THE UNIVERSITY *of* EDINBURGH

Restricting Information Flow in Security APIs via Typing

Gavin Keighren

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2014

Abstract

Security APIs are designed to enable the storage and processing of confidential data without that data becoming known to individuals who are not permitted to obtain it, and are central to the operation of Automated Teller Machines (ATM) networks, Electronic Point of Sale (EPOS) terminals, set-top boxes for subscription-based TV, pre-payment utility meters, and electronic ticketing for an increasing number of public transport systems (e.g., Oyster in London).

However, since the early 2000s, it has become clear that many of the security APIs in widespread use contain subtle flaws which allow malicious individuals to subvert the security restrictions and obtain confidential data that should be protected.

In this thesis, we attempt to address this problem by presenting a type system in which specific security properties are guaranteed to be enforced by security APIs that are well-typed. Since type-checking is a form of static analysis, it does not suffer from the scalability issues associated with approaches that simulate interactions between a security API and one or more malicious individuals.

We also show how our type system can be used to model an existing security API and provide the same guarantees of security that the API authors proved it upholds. This result follows directly from producing a well-typed implementation of the API, and demonstrates how our type system provides security guarantees without requiring additional API-specific proofs.

Lay Summary

Computers are used to protect an increasingly large amount of private information and also to control access to services. Common examples include ensuring that only you can withdraw money from your account at a cash machine, and that you can only watch the TV channels that you have paid for.

To provide these protections and controls, the computer system typically contains a physically secure component where sensitive ‘encryption keys’ and ‘decryption keys’ are stored. Encryption is a method which allows information to be made unreadable unless the corresponding ’decryption key’ is known. The secure component provides a series of actions which it can be asked to do by the computer. In the above cash machine example, this secure component is responsible for saying whether or not the PIN number entered is correct.

Unfortunately, it is very difficult to guarantee that the set of actions provided by the secure component cannot be used in an unintended way to bypass the protections that it is designed to enforce. The research presented in this thesis allows us to provide such a guarantee, for certain sets of actions. This result is achieved by providing rules which govern how an action should work. We then show how these rules can be applied to a set of actions which are known to be secure, and discuss how the restrictions enforced by the rules are similar to the restrictions that were required to make those actions secure.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Gavin Keighren*)

To all my friends and family who have supported me throughout, and to my supervisors who drove me forward and helped me through the difficult times. I could not have done it without each and every one of you.

Thank you.

Table of Contents

Chapter 1

Introduction

The transmission and processing of sensitive information forms a significant and ever-growing part of the digital world in which we live. In the consumer space, pay-TV is pervasive, online banking and shopping continue to go from strength to strength, while financial institutions are teaming up with retailers and mobile device manufacturers to try and convince us that we should use our mobile phones as electronic wallets. However, electronic payment systems such as pre-pay tokens for gas and electricity have been around since the 1970s, while many public transit systems around the world allow people to use pre-payment cards (e.g., the Oyster card in London).

The secure transmission of data itself is a goal which people have been striving to achieve for over 2,000 years. Developments in this area have generally been led by governments and militaries, and the resulting arms race between cryptographers and cryptanalysts is what has ultimately enabled so many of the services we use today. Cryptography is the foundation of information hiding, and modern ciphers such as Triple-DES and AES are considered unbreakable given current computing hardware. This is because the strength of such ciphers is related to the size of the encryption key and, unless some flaw is discovered that sufficiently weakens the cipher, it is enough to simply increase the key length in line with any increase in computing power.

Consequently, attempts to break secure systems typically focus on areas other than the cipher itself, with the aim of recovering the necessary encryption and/or decryption keys. These other areas include the *security protocol* which defines the sequence of messages that communicating parties exchange, and the *security API* which defines the functions that a security component provides. It is the latter of these two areas which we consider in this thesis.

1.1 Security APIs

A security API is a collection of operations which together implement an Application Programming Interface designed to enforce a given security policy through careful use of cryptographic operations. In general, the security policy will specify that certain data items cannot appear in the clear outside of the API code. Typically, security APIs are provided by a tamper-proof Hardware Security Module (HSM), but they may also exist as a software library.

A security API can be viewed as a barrier between the trusted and untrusted areas of a system. The goal of a security API is to provide the intended functionality while ensuring that no sensitive data can ever exist in the untrusted area, regardless of how calls to the API functions may be combined. A security API will typically take the form of a key management API, with additional functions determined by the functionality which the API is intended to provide. A key management API is a security API whose sole purpose is to store and allow the use of cryptographic keys, while ensuring that they do not become known to the applications which use the API.

Examples of security APIs include IBM's Common Cryptographic Architecture [CCA Basic Services Reference and Guide, Release 3.30] (a financial services API which includes functions for PIN verification) and RSA's PKCS #11 [PKCS #11] (a vendor-neutral standard providing many common features). Non-examples include TrueCrypt and IronKey — two systems which are used to encrypt sensitive information to prevent it from falling into the hands of others. These are not security APIs because they allow access to the sensitive data upon receipt of the correct password.

1.2 What is the "Problem"?

As noted in the previous section, the primary goal of a security API is to maintain the confidentiality of sensitive information, regardless of the sequence of API calls that a client application may make. A legitimate sequence of API calls which results in the discovery of sensitive information is known as a *security API attack* [Bond, 2004, (§3.1 ¶7–8)]. The "problem" is as follows:

> *For a given security API, how do we guarantee that no sequence of function calls allows confidential information to become known?*

Providing an answer to this question is the subject of this thesis.

1.2.1 Client-API Interaction Model

Although the focus of this thesis is on *security APIs*, it is necessary to consider how such APIs will be presented to clients and how those clients will interact with the API. The typical interaction model, which we follow, is shown in Fig. 1.1. In this model, multiple client programs interact with a single API, with all client programs sharing a common block of memory. The *low* memory is shared because our aim is to enforce the boundary between the *low* and *high* parts of the system, rather than to enforce any kind of separation on client data.

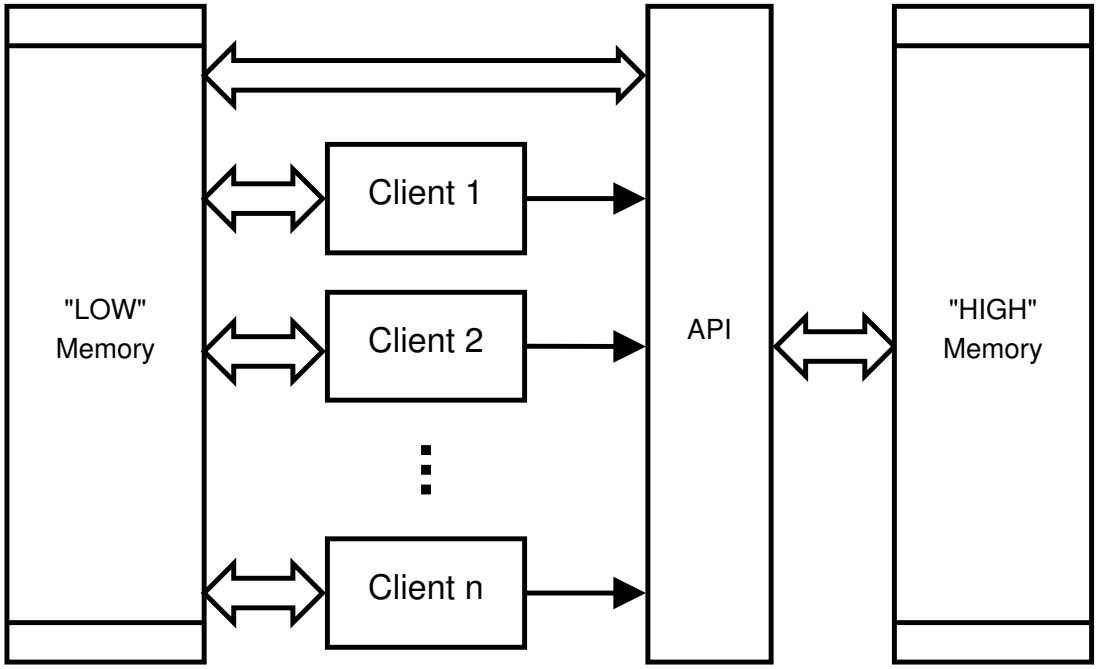

Figure 1.1: Client-API Interaction Model

The thick arrows represent permitted memory accesses: API code can access all memory locations, whereas the client programs may only access the public memory locations. This configuration follows from restrictions that exist in practice — with hardware security devices, the *high* memory is inside a tamper-proof enclosure, and with software APIs, access to the *high* memory locations is restricted by the compiler. A consequence of this view is that, when dealing with software APIs, the security properties only hold under the assumption that restricted memory locations are only accessed programmatically, i.e., there is no physical probing of the memory chips.

The thin arrows represent API function calls that may exist within client programs. For API functions which accept input parameters, a client program will be able to provide any data item that it has access to. In our model, we assume that a client

program has access to each and every data item that exists in the *low* part of the system. Consequently, the API is responsible for enforcing the boundary between the *high* and *low* parts of the system. We consider an API to be secure when the set of all data items in the *low* part of the system represents an upper bound on the set of terms that any client program is able to obtain information about. As such, our model assumes the worst-case scenario, where each client program has access to every single data item that it is permitted to. Our aim is to ensure that, even in this worst-case situation, a client program cannot learn anything about data in the *high* part of the system.

1.3 Contributions

The research presented in this thesis provides the following contributions to the field of security API analysis:

1. We provide a language and type system which enforces two desirable security properties on any API which can be represented and is well-typed. Previously, such a result would require API-specific proofs, while existing type-systems for security APIs enforce weaker security properties.

2. We present a well-typed implementation of a secure API [Cachin and Chandran, 2009] and show how our type system enforces similar restrictions and provides equivalent security guarantees. This can be viewed as both a validation of our type system, and of the restrictions enforced by that API.

1.4 Thesis Outline

Chapter 2 provides a background to the theory which our research builds upon, as well as an overview of similar work by others; Chapter 3 discusses the considerations and motivations that led to the design of our type system; Chapter 4 defines the formal system as well as the theorems which capture the security guarantees; Chapter 5 shows how our system can be applied to a key management API recently proposed by Cachin and Chandran, demonstrating how some of the security guarantees which they go to great lengths to prove follow directly from the use of our approach. Our conclusions are presented in Chapter 6, along with opportunities for future work. Full proofs of the lemmas and theorems from Chapter 4 are contained in the appendix.

Chapter 2

Background and Related Work

2.1 Multi-Level Security

Multi-level security (MLS) [Smith, 2006] is a method by which users of a system (i.e., people and processes) can be restricted in what elements of that system they may access (e.g., files, documents, services, etc.). Note that a user may also be an element, as is often the case with system processes. To this end, a MLS system associates a security label (or set of security labels) to each user and to each element. These labels are (partially) ordered so as to define a *security lattice*. Figure 2.1 shows the ordered security labels used by the UK Government [HMG Security Policy Framework].

The label associated with a user is termed their *access level*, and the label associated with an element is termed its *classification level*. Users can access elements when their access level is at least as high as the element's classification level. For example, using the security lattice shown in Fig. 2.1, a user whose clearance level is SECRET may access all elements except those with the classification level TOP SECRET; and elements whose classification level is SECRET may only be accessed by users with the clearance level SECRET or TOP SECRET.

It may also be the case that a MLS system has additional labels which are used to further restrict access on a *need-to-know* basis. For example, a US diplomat with CLASSIFIED clearance would not be able to read RESTRICTED documents marked with the additional label UK EYES ONLY. Incorporating such additional labels can be achieved by means of a more complex security lattice, and therefore remains within the scope of MLS. Figure 2.2 shows part of the semilattice corresponding to the UK GPMS augmented with the need-to-know labels UK (UK eyes only) and CANUKUS (Canadian, US and UK eyes only).

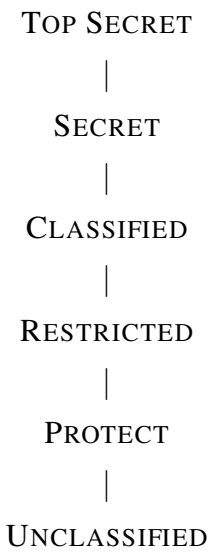

Figure 2.1: The UK Government Protective Marking System (GPMS)

2.1.1 Bell-LaPadula Security Model

The standard approach used to enforce multilevel security is the Bell-LaPadula (BLP) security model [Bell and LaPadula, 1973, 1976]. This model is concerned only with *confidentiality*, although similar models exist for other security properties (e.g., the Biba integrity model [Biba, 1977]). The BLP model comprises *subjects* and *objects* (what we termed "users" and "elements" above), each of which have an associated security level. The model also considers explicit need-to-know labels.

Two styles of system are dealt with: one where security classifications of objects, security clearances of subjects, and any need-to-know restrictions are all static; and another where access restrictions may be weakened (i.e., over time, the set of objects that a subject can access never decreases). The model does not deal with systems where access restrictions are strengthened. For both styles of system, Bell and LaPadula define the same basic security theorem, which can be stated informally as follows:

BLP Basic Security Theorem

Any system which starts in a secure state and which proceeds via security-preserving transitions will remain secure

The BLP model defines two mandatory security properties which must both hold for transitions to be necessarily security-preserving. These properties are known as the *simple security property* and the $*$*-security property* (read "star security property"), and are defined informally as follows:

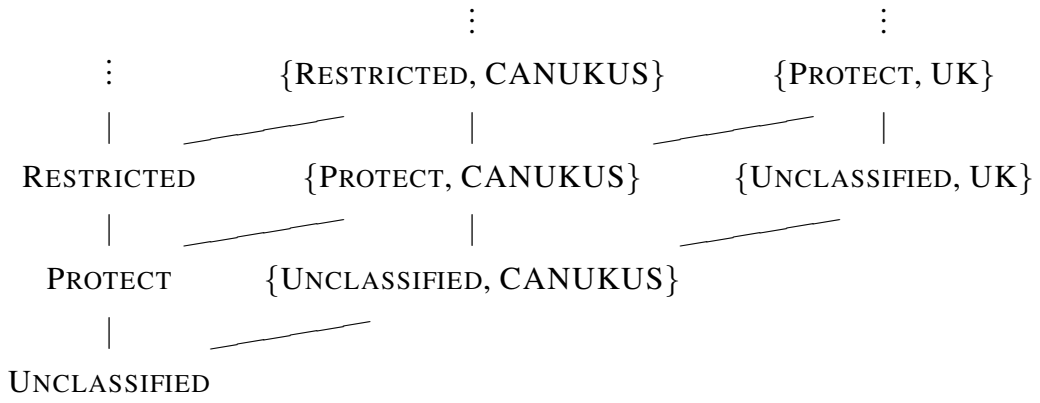

Figure 2.2: Part of the security lattice produced by augmenting the UK Government Protective Marking System (GPMS) with need-to-know labels

BLP Simple Security Property

A subject can observe *an object only when the subject's clearance level is at least as high as the object's classification level*

BLP ⋆-Security Property

A subject can modify *an object only when the subject's clearance level is equal to or lower than the object's classification level*

The simple security property is often termed "no read up" and the $*$-security property is often termed "no write down".

In practice however, it is common for a subject with a high clearance level to want to modify a low security object based solely upon data from other low security objects. The BLP model permits such an operation to be carried out only by subjects with a low clearance level, but by temporarily downgrading the subject's clearance level, the desired operations can be carried out. Of course, it is critical that any modifications to the low security objects are indeed based solely upon low security data.

2.2 Information Flow

Information flow captures the relationship between data and the operations that are carried out on that data. The operations generally include those which define the logical control flow of a program, as well as the read and write operations which the BLP model constrains. An insecure information flow exists when a particular piece of data is dependent upon other, higher-confidentiality data.

Information flow was originally formalised by Denning [Denning, 1975, 1976], providing a mechanism for identifying and thus preventing insecure information flows which arise from the logical execution of a program (i.e., conditional statements, reads and writes). Flows which may arise from covert channels such as power usage, system load, execution time, etc. are ignored.

There are two types of information flow: *explicit* and *implicit*. Explicit flows are those in which a subject directly observes or modifies an object when they do not have the necessary permissions. Implicit flows arise when the outcome of some operation can be determined by code which has insufficient clearance to observe objects of that sensitivity. For example:

```
if (high == 1) then
  low := 1
else
  low := 0
```

This conditional statement leaks information about the confidential variable *high* into the non-confidential variable *low*, even though no direct write-downs occur.

The security properties defined by the BLP model do actually prevent the implicit flow which is shown in the above example, even though the definitions only talk about restricting explicit flows. This is because the code may be permitted to read the value of *high*, or modify the value of *low*, but not both. As a result, no clearance level exists such that the program will satisfy both security properties in the BLP model.

Now consider the following example, where the clearance level of the program is high:

```
if (low == 1) then
  low' := 1
else
  low' := 0
```

With a high clearance level, this program snippet contravenes the $*$-security property, and is therefore deemed insecure. As noted at the end of §2.1.1, we can avoid this problem by allowing programs to temporarily reduce their clearance level. However, we can also apply the same trick when executing the first example — temporarily reduce the program's clearance level in each of the branches. But this first example contains an implicit information flow, therefore we must find a way to differentiate between these two cases.

Fenton's Data Mark Machine [Fenton, 1974] was proposed as a mechanism for preventing implicit flows arising from program control-flow. The context in which an expression is being evaluated is assigned a security level, and this is used to control what operations that expression is permitted to do. Specifically, the security level of the context is the minimum level to which information may flow when evaluating any of the subsequent branch expressions. When evaluation of the guard expression in a conditional statement has completed, the security level of the context becomes the least upper bound of its current level and that of the guard. In the first example given above, the confidentiality level of the guard expression is high because it involves the confidential variable *high*. As a result, to guarantee that no insecure information flows exist, assignments to non-confidential variables cannot appear in either branch. Since both branches contain such operations, the code snippet is insecure. In the second example, the confidentiality level of the guard expression is low because it involves the non-confidential variable *low* and the non-confidential constant 1. Since neither branch includes assignments to variables with a confidentiality level less than that of the guard expression, the code snippet is (correctly) deemed to be secure.

A detailed overview of the field of information-flow security from the early 1970s to the early 2000s is given in [Sabelfeld and Myers, 2003], focusing on research which makes use of static analysis methods.

2.2.1 Non-Interference

Introduced by Goguen and Meseguer [Goguen and Meseguer, 1982], *non-interference* is a more general security property that implies a lack of information flows from high confidentiality data to low confidentiality data. It was originally stated in terms of the interactions between different users of a system:

Goguen and Meseguer's General Statement of Non-Interference

> One group of users, using a certain set of commands, is *non-interfering* with another group of users if what the first group does with those commands has no effect on what the second group of users can see

In the context of programs, non-interference means that confidential inputs cannot influence public outputs. This is often generalised so that the program inputs include the initial values of any program variables, and the outputs include the final values of any program variables. Additionally, values created on-the-fly during the program's

execution are considered to be part of its inputs (all such values are assumed to be generated in advance, then provided as input to the program which simply picks a fresh one at the appropriate point during its execution).

2.3 Security Analyses of Cryptographic Systems

Approaches to analysing the security of cryptographic systems fall into one of two camps: the first one uses a *symbolic* model of cryptography (as we will do), while the other uses a *computational* model of cryptography. The symbolic model typically assumes that encryption is *perfect* — that is, given a particular piece of ciphertext, no information can be obtained about the plaintext that it contains without access to the correct decryption key. In practice however, this assumption may not be valid [Pereira and Quisquater, 2000].

The computational and symbolic analysis approaches were bridged by Abadi and Rogaway [Abadi and Rogaway, 2000] who proved that, with the following restrictions placed upon the encryption scheme, security in the symbolic model implies security in the computational model:

1. The cipher must be *repetition concealing*: given any two non-equal ciphertexts, it must be computationally infeasible to determine whether the same plain-text was encrypted

2. The cipher must be *which-key concealing*: given some piece of ciphertext, it must be computationally infeasible to determine which key was used

3. The cipher must be *message-length concealing*: given any two ciphertexts which are non-equal, it must be computationally infeasible to determine whether or not the encrypted plain-texts are of the same length

4. There must be *no key-cycles*: no key can encrypt a message or ciphertext which required that key to produce

Abadi and Rogaway termed a cipher which possesses the first three properties *type-0*, and show how ciphers which do not directly possess these properties can be used in such a way so that the encryption scheme does.

Abadi and Rogaway achieve their main result by introducing the notion of *patterns*: an expression that may contain parts which the adversary cannot decrypt. A pattern

is obtained from a normal expression by replacing all ciphertexts that the adversary cannot decrypt with $\Box$. From the perspective of the adversary, two expressions are the same when they reduce to the same pattern. That is, any differences that do exist are hidden from the adversary. Consider the following example, where $\{m\}_k$ denotes the encryption of the message m under the key k:

$$\mathcal{K}_{adv} ::= k_1 \qquad \textit{keys known to the adversary}$$

$$\begin{array}{lllll} (1) & \{\{m\}_{k_2}\}_{k_1} & & \{\Box\}_{k_1} \\ (2) & \{\{m\}_{k_1}\}_{k_2} & \rightsquigarrow & \Box \\ (3) & \{m\}_{k_2} & & \Box \end{array}$$

The adversary cannot decrypt the second or third ciphertext and is therefore unable to determine that those expressions are different from each other. However, the adversary can decrypt the first ciphertext and thus determine that it is different from either of the other two.

2.4 Type Systems

Type systems are a form of static analysis which guarantee that specific properties hold of programs which are well-typed. In this thesis, we show how non-interference can be enforced via a type system for programs that make use of certain cryptographic primitives. Our type system builds upon previous work carried out by Dennis Volpano, Geoffrey Smith and Cynthia Irvine [Volpano et al., 1996, Volpano and Smith, 1997], and Eijiro Sumii and Benjamin Pierce [Sumii and Pierce, 2003], amongst others. A comprehensive overview (up to 2003) of language-based approaches to information-flow analysis is given in [Sabelfeld and Myers, 2003].

2.4.1 Non-Interference in a Simple Imperative Language

Volpano, Smith and Irvine [Volpano et al., 1996] were first to formulate a type system that would guarantee non-interference for programs specified in a simple imperative language with conditional branching. The type system was based upon Denning's approach [Denning, 1975, 1976] and also demonstrated the soundness of Denning's rules. The type system was later extended to include support for first-order procedure calls [Volpano and Smith, 1997].

x		*Variables*
n		*Integers*
a		*Locations*
p	$::= e \mid c$	*Phrases*
e	$::= x \mid n \mid a \mid e + e' \mid e - e' \mid e = e' \mid e < e' \mid$ proc(in x_1, inout x_2, out x_3) c	*Expressions*
c	$::= e := e' \mid e ; e' \mid e(e_1, e_2, e_3) \mid$ while e do $c \mid$ if e then c else $c' \mid$ letvar $x := e$ in $c \mid$ letproc x(in x_1, inout x_2, out x_3) c in c'	*Commands*
τ	$::= l$	*Data types*
π	$::= \tau \mid \tau\ proc(\tau_1, \tau_2\ var, \tau_3\ acc) \mid \tau\ cmd$	*Procedure types*
ρ	$::= \pi \mid \tau\ var \mid \tau\ acc$	*Phrase types*

Figure 2.3: Syntax of Volpano, Smith and Irvine's extended type system, from [Volpano and Smith, 1997][1]

Figure 2.3 presents the syntax for the extended version of Volpano, Smith and Irvine's type system.[1] Typing judgements and evaluation relations, respectively, have the form:

$$\lambda; \gamma \vdash p : \rho \qquad \mu \vdash p \Rightarrow \mu'$$

where λ is a mapping from locations to types, γ is a mapping from variables to types, and μ is a store. The maximum security level of values which may be stored in a location a and variable x, respectively, are denoted as follows:

$$\lambda(a) \qquad \gamma(x)$$

They then prove a number of properties for well-typed terms. From a security perspective, the important ones are as follows:

Volpano, Smith and Irvine's Simple Security Lemma

If $\lambda; \gamma \vdash e : \tau$ then, for every a in e, $\lambda(a) \leq \tau$, and for every x free in e, $\gamma(x) \leq \tau$

[1] In the original presentation of the type system, locations are represented by l and security levels by s. This has been changed here to match the notation we use in our type system.

Volpano, Smith and Irvine's Confinement Lemma

If $\lambda \vdash c : \tau\ cmd$, $\mu \vdash c \Rightarrow \mu'$, $\mathsf{dom}(\lambda) = \mathsf{dom}(\mu)$ and a is a location assigned to in c, then either $\tau \leq \lambda(a)$ or $\mu(a) = \mu'(a)$

Volpano, Smith and Irvine's Non-Interference Theorem

If $\lambda \vdash c : \rho$, $\mu_1 \vdash c \Rightarrow \mu_1'$, $\mu_2 \vdash c \Rightarrow \mu_2'$ and $\mathsf{dom}(\lambda) = \mathsf{dom}(\mu_1) = \mathsf{dom}(\mu_2)$ then, for all a such that $\lambda(a) \leq \tau$, $\mu_1(a) = \mu_2(a)$ implies $\mu_1'(a) = \mu_2'(a)$

The simple security lemma corresponds exactly with the simple security property from the Bell-LaPadula model, and the confinement lemma corresponds to the *-property. The non-interference theorem is a formal interpretation of Goguen and Meseguer's general statement (see §2.2.1).

2.4.2 Non-Interference and Cryptography

In [Sumii and Pierce, 2003], the authors present an extension to the simply typed lambda calculus [Church, 1940] that includes primitives for encryption, decryption and key generation, where encryption and decryption are assumed to be symmetric and perfect. The syntax is shown in Fig. 2.4. Note that a decryption operation is followed by one of two possible expressions — the first is for when the decryption succeeds, and the latter is for when it fails. Consequently, there exists the implicit assumption that it is possible to determine whether or not a decryption has failed.

The types present in the simply typed λ-calculus are extended with $\mathsf{key}[\tau]$ and $\mathsf{bits}[\tau]$ — types for keys and ciphertexts, respectively. The type $\mathsf{key}[\tau]$ is given to keys which may encrypt expressions of type τ, and the type $\mathsf{bits}[\tau]$ is given to ciphertext containing a plain-text expression of type τ.

To prove the security properties of well-typed expressions, Sumii and Pierce define a relation such that two related terms are behaviourally equivalent from the perspective of the adversary. In other words, the adversary is unable to distinguish between two related terms.

Sumii and Pierce's Basic Logical Relation

- Two integers are related if and only if they are equal
- Two functions are related if and only if they return related output when called with related inputs

i	*Integers*	k	*Keys*	x	*Variables*	e	*Expressions*

$int\_op_n(e_1, \ldots, e_n)$	*Integer arithmetic operations*
$\lambda x.e$	*Function definition*
$e_1 e_2$	*Function application*
$\langle e_1, \ldots, e_n \rangle$	*Tuples*
$\#_i(e)$	*i^{th} element of a tuple*
$\mathsf{in}_i(e)$	*i^{th} element of a disjoint sum*
$\mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1(x_1) \Rightarrow e_1 \,[\!]\, \ldots \,[\!]\, \mathsf{in}_n(x_n) \Rightarrow e_n$	*Case split*
$\mathsf{new}\ x\ \mathsf{in}\ e$	*Key generation*
$\{e_1\}_{e_2}$	*Encryption*
$\mathsf{let}\ \{x\}_{e_1} = e_2\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4$	*Decryption*

$\tau ::= \mathsf{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \cdots \times \tau_2 \mid \tau_1 + \cdots + \tau_2 \mid \mathsf{key}[\tau] \mid \mathsf{bits}[\tau]$ *Types*

Figure 2.4: Syntax of Sumii and Pierce's cryptographic λ-calculus, from [Sumii and Pierce, 2003]

- Two tuples are related if and only if they are of the same length and each pair of associated elements are related
- Two tagged values are related if and only if their tags are equal and their bodies are related
- Two keys are related if and only if they are identical and public
- Two ciphertexts are related if and only if the same encryption key was used to generate them, and either:
  - the key is secret and is permitted to encrypt both plaintexts,[2] or else
  - the key is not secret and the two plaintexts are related

Given two copies of a particular program which differ only in the values of some secret terms, that program preserves the confidentiality of those terms if and only if the two copies are related to each other. In other words, the adversary is unable to distinguish between the public outputs from the two instances of the program.

[2]The intent here is to ensure that the adversary cannot alter the set of values that a secret key will be used to encrypt

2.5 Indistinguishability

Sumii and Pierce's definition for their basic logical relation states that two ciphertexts are related if the keys used to create them are identical and both plaintexts may be encrypted by that key.[2] Implicit in this definition is the assumption that it is impossible to determine the contents of some piece of ciphertext without knowing the key. This property is known as *repetition concealing* (see [Abadi and Rogaway, 2000, §4.2] for a brief overview) and is possessed by many common ciphers (e.g., DES in CBC and CTR mode [Bellare et al., 1997]).

However, encryption breaks non-interference since ciphertext is dependent upon the specific key and message used — changing the key or message alters the ciphertext. To resolve this problem, various researchers (e.g., [Backes and Pfitzmann, 2002], [Laud, 2001]) have utilised *possibilistic non-interference* [McCullough, 1988]. Informally, possibilistic non-interference states that two non-equal values can be considered equal if the adversary is unable to determine that they correspond to distinct terms. In relation to cryptographic operations, two non-equal ciphertexts can be considered equal if the adversary is unable to determine that either a different encryption key has been used, or a different plaintext has been encrypted. It therefore follows that it is necessary for the cipher to be probabilistic.[3] For purely deterministic ciphers, it is necessary to use Goguen and Meseguer's more strict definition where ciphertexts are compared using bit-wise or structural equality.

Assuming that the cipher is repetition concealing and which-key concealing, two ciphertexts of equal length can be distinguished in the following ways:

- If both ciphertexts can successfully be decrypted and the resulting plaintexts can be distinguished
- If some key or procedure is known to correctly decrypt one ciphertext but not the other

In cases where neither ciphertext can be decrypted and the plaintexts obtained, there is no way to determine the existence of any differences and thus they must be considered indistinguishable.

[3] Deterministic ciphers which first combine the message with some unique random value (i.e., the initialisation vector) can be considered probabilistic since multiple encryptions with the same key and message will produce different ciphertexts.

2.5.1 Cryptographically-Masked Flows

Cryptographically-masked flows [Askarov et al., 2008] are semantically-equivalent to the above property, and the authors provide a type system which enforces this property for programs composed from a small imperative language with cryptographic primitives. The type system shares a number of similarities with the research presented in this thesis but, because the type system is targeted at security protocols, it does not deal with function application. As we will demonstrate, API functions can be viewed as the boundary between secure and insecure domains, across which no sensitive data should flow. In [Askarov et al., 2008], this boundary is determined by the incoming and outgoing channels from the communicating processes. Like much of the research in this field, the authors consider only two security levels, and make a few simplifications based on this representation. This representation does not alter the validity of the security proofs, but it does introduce an additional abstraction level when the system being modelled comprises more than two security levels.

The computational soundness of this approach was shown to hold [Laud, 2008], provided that the actual bit-string of a ciphertext is never used, and that no public result is dependent upon the value of any secret encryption key. Indeed, these are two restrictions which our type system is also required to enforce.

2.6 Contributions Revisited

Our primary contribution is a type system for expressions (i.e., programs) that include calls to functions from some security API, and we prove that well-typed expressions are *non-interferent*. We formally define this property in Definition 4.11, but it can be stated informally as follows:

Non-Interferent Expression

> An expression is *non-interferent* at some security level l when the execution of that expression does not reveal any information about values whose security level is higher than l

In other words, there are no information flows to l from higher security levels.

In Chapter 3, we describe and justify the components of our formal model as presented in Chapter 4, then discuss the two security properties that our type system will enforce: *confidentiality* and *non-interference*. The former property is implied by the latter, but

is included explicitly as a means to introduce certain aspects of our type system more easily. Our proof that a well-typed expression is non-interferent involves simulating two related executions of that expression, where the only differences between each execution is in those initial values which the adversary cannot obtain directly. We then show that the adversary is incapable of distinguishing the two values produced. Consequently, it is critical that the capabilities we infer on the adversary to distinguish between any two values are realistic. §3.4 discusses these capabilities, which serves to justify the formal definition of indistinguishability that we give in Definition 4.8.

In Chapter 4, we define an evaluation semantics for a simple imperative language with primitives for symmetric encryption and decryption. The evaluation semantics are intended to be very intuitive, to ensure that the power of our type system comes from the type rules themselves, rather than from any clever program semantics.

We formally define what it means for an expression to be *confidentiality-preserving* and *non-interferent*, in Definition 4.7 and Definition 4.11, respectively. Only once we have these formal definitions, do we introduce our typing rules.

Our overall type system is influenced by a number of existing type systems for analysing information-flow, e.g., [Volpano et al., 1996], [Volpano and Smith, 1997], [Sumii and Pierce, 2001]. Differences between these systems and ours arise from the adversary model that we use — since we are considering security APIs, the adversary may be able to execute cryptographic operations where he does not have access to the key nor the message.

Our second contribution is a demonstration that our type system can model practical security APIs, imparting upon them the enforced security properties. In Chapter 5, we present a well-typed implementation of the security API proposed by Cachin and Chandran [Cachin and Chandran, 2009]. Consequently, the model is guaranteed to be *non-interferent* without requiring additional proofs specific to that API. Our research can therefore be viewed as a validation of the restrictions that Cachin and Chandran enforce, in an API-agnostic manner.

One feature of our type system which simplifies the modelling process is that the type rules are parameterised over any security level lattice which has well-defined top and bottom elements. In contrast, security type systems typically use a security lattice comprising only two levels ($high > low$), simplifying the theory but complicating any application of those rules — you first have to determine the minimum security level of values which are deemed confidential, then map all levels below that to $\perp$ and all

levels greater than or equal to it to $\top$. If you want to verify security properties across three or more levels, you will need to create multiple models.

2.7 Related Work

Security APIs are a fairly recent target for security researchers, e.g., [Anderson, 2000], [Clulow, 2003], [Bond, 2004], [Youn, 2004], [Keighren, 2006], [Tsalapati, 2007], with formal analyses often building upon existing methods for analysing security protocols. However, while security protocols are generally stateless and define a fixed and finite sequence of messages to exchange, security APIs are typically stateful and comprise many functions, which may be called in any order and as often as desired. A recent overview of the field of security API analysis is given in [Focardi et al., 2011].

2.7.1 Bengtson, Bhargavan, Fournet, Gordon and Maffeis

Bengtson et al. [Bengtson et al., 2008] present a type-checker for verifying security properties of cryptographic protocols and access control systems written in F#. The target code is annotated with *refinement types* that specify properties that should hold. The theory behind the type-checker is presented as a form of the λ-calculus.

Refinement types are of the form $\{x{:}T \mid C\}$, where C is a first-order formula whose free variables are a subset of $\{x\}$. A value v may be assigned this type when v has type T and C holds when v is substituted for x in C. Refinements types are more powerful than those in our type system, but due to the presence of first-order formulae, the type-checking process may be undecidable.

The authors focus on authentication and authorisation properties, but their type system is not limited to these areas. The specification and verification of first-order formulae is done via **assume** and **assert** operations, respectively. All cryptographic operations are handled implicitly via the abstract notion of *sealing* and *unsealing*.

The authors goal is to verify that an application is *robustly safe*. This means that, even in the presence of an arbitrary adversary, none of the **assert** statements in the application will fail.

2.7.2 Buttyán and Thong

Buttyán and Thong [Buttyán and Thong, 2008] analyse four commands from the API for the VISA Security Module using the spi calculus [Abadi and Gordon, 1998] — an

extension of the π-calculus [Milner et al., 1992] which includes primitives for shared-key encryption and decryption. The π-calculus is a process calculi and therefore has features for modelling concurrent systems. Consequently, the spi calculus has been used to analyse security protocols [Abadi, 1999].

In the spi calculus, confidentiality can be couched in the following terms. Given a process *P(x)*, where x is some sensitive term that should remain secret, and an adversary process Q, the input x is not leaked if Q cannot distinguish between the cases where it is running in parallel with *P(m)* and the cases where it is running in parallel with *P(n)*, for all m and n. The confidentiality of x is protected through the use of dynamically generated *Secret* terms (e.g., keys and nonces). Dynamic generation of some secret terms is a necessary requirement since probabilistic encryption is modelled via the inclusion of a random *confounder* in the plaintext message. Without confounders, two ciphertexts generated from the same key and message pair would be equal and thus the adversary could easily determine whether or not two ciphertexts contained the same message. Buttyán and Thong encode a security API as follows:

- Each API function is modelled as a separate process
- A distinct public communication channel exists for each API function (i.e., the parameters of a function call are sent on a dedicated channel).
- All output from API calls are sent on a single public channel, c_{user}
- The API comprises the parallel composition of the infinite replication of the function processes
- The adversary's initial knowledge is output on c_{user} immediately before the API process
- Ciphertexts contain the plaintext message as well as a "tag" denoting the message type

Buttyán and Thong are forced to use tagged ciphertexts because key types in their system do not include the type of data on which a key may be used, and because messages sent to the API cannot be uniquely determined on structure alone (if they could, then there would be no need for a separate channel to be associated with each API function).

2.7.3 Centenaro, Focardi, Luccio and Steel

Centenaro et al. [Centenaro et al., 2009] present a type system to prove the security of PIN processing APIs, which naturally must declassify some information about the value of the confidential PIN (i.e., whether or not the input PIN matches the actual PIN). The authors' main result is that well-typed API functions are *robust* [Myers et al., 2006] and, if no declassification is performed, *non-interferent*. Robustness is a weakening of non-interference that allows for declassification, but only where the adversary cannot influence what is declassified.

The enforcement of robustness is achieved through the use of MACs (Message Authentication Codes) which prevent the adversary from being able to tamper with the function input parameters. Since the adversary cannot modify individual input parameters, he cannot affect whether declassification occurs, nor affect what values are declassified.

Similarly to our research and that of others, the type for cryptographic keys includes the maximum security level of the data which that key may be used to encrypt, and the type of ciphertexts includes the type of the plaintext within. However, the type system does not deal with decryption failure — such evaluations are defined to be non-terminating. This removes a valuable side-channel that the adversary may use to distinguish between two ciphertexts, and one which we include in our research.

2.7.4 Centenaro, Focardi and Luccio

Centenaro, Focardi and Luccio [Centenaro et al., 2012] provide a type system for analysing the key management operations of PKCS #11. The type system makes a distinction between key wrapping/unwrapping and regular encryption/decryption, as well as having rules for deriving new keys from existing keys. The authors prove that their type system enforces two security properties:

1. A sensitive value will not become known by the adversary as a result of executing well-typed API functions

2. An *always-sensitive* key is never known by the adversary, nor will any new key derived from an always-sensitive key become known by the adversary

The first property corresponds to confidentiality and is explicitly mentioned in the PKCS #11 documentation, while the latter corresponds to integrity for those values which are considered to be high-integrity.

To enforce these security properties, the type system requires that keys are not used for conflicting operations (e.g., key wrapping and arbitrary decryption). In other words, permitted combinations of roles in PKCS #11 are mapped to distinct key types which are proven to enforce the desired security properties. The type system that we present in this thesis is intended to provide similar restrictions, but where the various key types are defined by whatever security level lattice is used.

In addition to achieving the security result via enforcing restrictions on the usage of each key, the authors also present a novel solution whereby a new key is produced for each type of use, derived from the original key using some sensitive seed value. This results in the same separation of use, but done in a way that is transparent to the end user of the API. It should be noted however that the derivation mechanism must be secure, so as to avoid attacks which convert keys to a different type, such as has been shown to exist with IBM's CCA API [Bond, 2000].

Chapter 3

Development of K

In this chapter, we build up an informal description of our type system — called K — and in doing so, provide justification for the design decisions that have been made in the course of its development. This will serve as a system primer for the reader, before the formal definitions and properties are presented in the next chapter.

3.1 System Framework

The primary aim of our research is to provide a mechanism for ensuring the absence of insecure information flows in security API implementations, as well as programs that include calls to those functions. We do this by defining a syntax and accompanying type system for both API implementations and client programs, such that well-typed instances are guaranteed to preserve the confidentiality of sensitive data and be free from insecure information flows arising as a result of the logical program control flow.

We restrict ourselves to well-typed client programs for two reasons. The first is that it allows us to focus on preventing API designers from creating an API that leaks sensitive information even when used correctly. The second is that there are many scenarios where the client code will be well-typed. Examples of the latter are software APIs where client code is run through a compiler than can enforce typing, or where the API code itself is able to determine at runtime what type a function input has (e.g., via cryptographically-bound tagging).

By considering only well-typed programs, our type system may not prevent attacks that rely on type-confusion. That is, where sensitive data is leaked as a result of API functions being called with parameters which are of a different type than expected. Section 4.6 gives an example of such attacks and discusses this problem in more detail.

3.1.1 Data Model

In K, data items are termed *values*, as they are a canonical representation of some particular piece of information. A value may be *primitive* or *compound* — a primitive value is the most basic form of value, whereas a compound value is built up from other compound or primitive values.

The primitive values that we include in K are as follows:

- **Boolean Constants:** TRUE, FALSE
  These represent fixed bit-strings which denote the Boolean values TRUE and FALSE, respectively.

- **Arbitrary Constants:** c
  These represent fixed bit-strings which cannot be used to encrypt other values.

- **Cryptographic Keys:** k
  These represent fixed bit-strings which can be used with a symmetric cipher to encrypt other values, and decrypt ciphertext.

- **Location Identifiers:** a
  These represent the address of a memory location which stores some particular value. Knowledge of a location identifier does not imply knowledge of, or ability to obtain, the value which is currently stored in that location.

We also include the following compound values:

- **Ciphertext:** $\text{ctxt}(n, k, v)$
  These represent the result of encrypting the (primitive or compound) value v with the key k. The encryption algorithm is required to be a type-0 symmetric cipher, and we represent the probabilistic nature of such ciphers by including a fresh *confounder* n in each ciphertext produced.

We do not include numbers in K, therefore the only comparison operation which may be carried out between two values is a bit-wise equality test. In our abstract semantics, this corresponds to structural equivalence.

3.1.2 Code Phrases

In K, both client code and API code are drawn from the same set of code phrases, termed *expressions*. We believe that placing artificial restrictions upon either client

code or API code would limit the accuracy and applicability of our approach. Instead, differences between client and API code arise solely from the access rights assigned to each, i.e., API code executes with greater privileges than client code.

The expressions that client code and API code are drawn from are as follows:

- **Symmetric Encryption:** $\text{senc}(e_k, e_m)$
  If e_k evaluates to a key k, and e_m evaluates to a value v, then $\text{senc}(e_k, e_m)$ evaluates to the ciphertext $\text{ctxt}(n, k, v)$, where n is a fresh *confounder* (random value). This represents the probabilistic encryption of v with k. Cryptographic operations are discussed in more detail in §3.2.

- **Symmetric Decryption:** $\text{try } \text{sdec}(e_k, e_c) = x \text{ in } e_1 \text{ else } e_2$
  First, e_k and e_c are fully evaluated, producing v_k and v_c, respectively. If v_k is a key and v_c is ciphertext of the form $\text{ctxt}(n, v_k, v)$, v is substituted for x in e_1, which is subsequently evaluated. Otherwise, e_2 is evaluated (with no substitution carried out). We therefore require that the cryptographic algorithm is able to detect when a decryption operation fails. The reasons for the decryption operation being of this form are discussed further in §3.1.3.

- **Function Calls:** $f(e_1, \ldots, e_n)$
  Function calls are the sole means by which client code interfaces with API code. Functions may be internal to the API, callable by client code, or internal to the client code. From a security perspective, the only differences between these forms are the security levels of the input and output values, and the permissions with which the function body executes. For example, client functions will be executed with fewer permissions and access rights than API functions.

- **Scoped Variable Declaration:** $\text{let } x = e_1 \text{ in } e_2$
  First, e_1 is fully evaluated, producing the value v. Next, v is substituted for x in the expression e_2, which is subsequently evaluated to give the overall result.

- **Conditional Branching:** $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$
  First, e_1 is fully evaluated, producing v_1. If v_1 is TRUE then e_2 is subsequently evaluated, otherwise e_3 is subsequently evaluated.

- **Equality Testing:** $e_1 == e_2$
  After e_1 and e_2 have been fully evaluated, they are compared using structural equality, corresponding to bit-wise equality in practice. If they are equal, this expression evaluates to TRUE, otherwise it evaluates to FALSE.

- **Assignment:** $e_1 := e_2$

  If e_1 evaluates to a location identifier, then the associated location is updated to contain the value to which e_2 evaluates, v. The overall expression evaluates to v.

- **Dereferencing:** $*e$

  If e evaluates to a location identifier, then $*e$ evaluates to the value currently stored in that location. To simplify our analysis slightly, we require that location are initialised with some default value to ensure that the result of any dereference operation is well defined.

3.1.3 Decryption Failure

The symmetric decryption expression directly handles the case when the decryption key is incorrect — a similar approach to that taken in other systems that deal with cryptography, such as [Sumii and Pierce, 2001]. One reason for doing this is that the success or otherwise of a decryption operation cannot be determined in advance since the choice of decryption key and/or ciphertext may be controlled by the adversary. However, this does necessitate that the errors are handled immediately; we could have used a general error-handling expression similar to the try-catch block common in programming languages such as Java. Consider the following example, where v_k is some confidential key and a corresponds to some public location:

```
try
   let x = (b := a) in
   let y = sdec(v_k, v_c) in
      a := TRUE
catch INCORRECT_KEY
   a := FALSE
```

After this snippet of code has been executed, a will contain the value TRUE if and only if the decryption operation succeeded. Consequently, there is an information flow from the *high* part of the system (containing the confidential key v_k) to the *low* part of the system (containing the public location a).

To prevent this indirect information flow, it is necessary to prevent any assignment to *low* locations anywhere within the try-catch statement after the decryption. This restriction is cleaner and more explicit if the above code snippet must be written as follows:

let $x = (b := a)$ in
  try $\mathrm{sdec}(v_k, v_c) = y$ in
    $a :=$ TRUE
  else
    $a :=$ FALSE

The try-catch construct does not confer any additional expressivity on the language, but rather helps to simplify exception-throwing and exception-handling code. Including the try-catch construct in K would serve only to complicate the proofs. For this reason, we instead require the decryption operation to handle failed decryption immediately.

3.1.4 Programs

The main aspect of our interaction model is the distinction between privileged API code (running inside a tamper-proof hardware security module, or with higher system privileges) and restricted client code (running on any other device, or with basic system privileges). A *program* comprises client code plus the API code for each corresponding function call that exists in the client code, as shown in Fig. 3.1. Consequently, the overall execution trace will switch between restricted and privileged blocks of code whenever there is a call to an API function. With respect to confidentiality, an insecure information flow will exist if the data returned by any API function call allows the client to discover information about a sensitive data item. We aim to prevent such insecure information flows from arising, thus preserving the invariant that the client program only has information about data in the *low* part of the system. A minimum requirement for this is that any API function which may be called by client code should not return sensitive data items directly.

As defined by our interaction model, client code may directly access any data item in the *low* part of the system, and API code may directly access data items in both the *low* and *high* parts of the system.[1] Similarly to before, these access assumptions are upper bounds on what will happen in practice. This follows from our interaction model, where client code is prevented from directly accessing data items in the *high* part of the system.

Both client code and API code may contain recursive function calls, therefore it is possible for a program execution to never terminate. The potential for non-termination

[1] For data items corresponding to ciphertext, the client or API code may not necessarily be able to obtain the data item which has been encrypted.

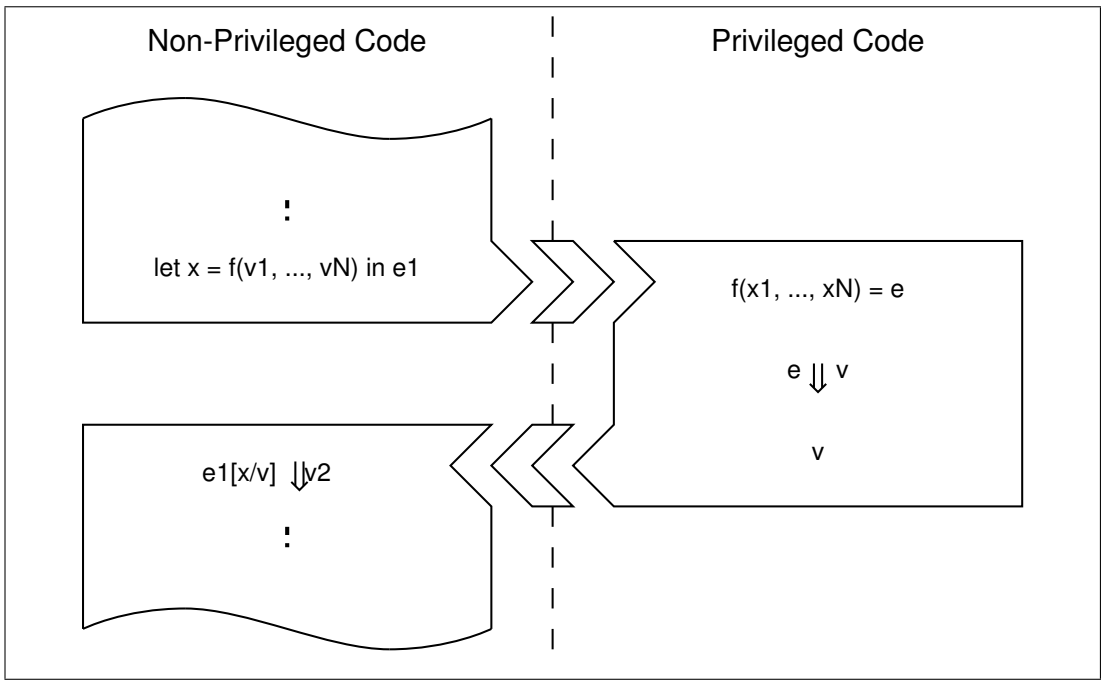

Figure 3.1: Overview of a Program in our Interaction Model

introduces a side-channel which may give rise to insecure information flows (e.g., if termination is dependent upon the value of some sensitive data item). We ignore this particular side-channel, focusing instead on side-channels which arise as a result of program flow (i.e., conditional branching).

3.1.5 Richer Security Lattices

So far, we have only considered the partitioning of data items into two parts. The formal system that will be presented in the next chapter is defined with respect to an arbitrary security lattice. A richer lattice allows for restrictions to be enforced beyond those which are necessary to ensure the fundamental security properties that we want. For example, we can enforce user-based restrictions on keys, thereby preventing one client from being able to encrypt the secret keys belonging to another client. Such restrictions could be achieved as follows:

- Security levels include a component corresponding to a set of users who are permitted to obtain the associated data item. For example, (H, {alice,bob})
- The security level (l_1, U_1) is dominated by the security level (l_2, U_2) iff $l_1 \leq l_2$ and $U_2 \subseteq U_1$. For example, (H, {bob}) dominates (L, {alice,bob}), while (L, {alice}) and (L, {bob}) are incomparable.

- The clearance level of user u is $(l_u, \{u\})$.

PKCS #11 [PKCS #11] is an example of a real-world security API where a richer lattice can enforce additional properties that the API is designed to uphold. In PKCS #11 it should not be possible to encrypt keys marked as *unextractable*. This means that sensitive keys are split into two sub-groups: those which can be encrypted and those which cannot. This naturally lends itself to using a security lattice comprising three levels.

In Chapter 5, when we apply our type system to the security API presented in [Cachin and Chandran, 2009], we employ a security lattice comparable to the one outlined above.

3.2 Cryptographic Operations

We consider two basic cryptographic operations in our system: symmetric encryption and symmetric decryption. Other operations such as public key cryptography, digital signatures, and key derivation are left as future work. We require certain standard properties of the symmetric cryptographic operations:

1. The cipher is which-key concealing, repetition concealing, and message-length concealing (i.e., a type-0 cipher)

2. Decryption failure can be detected

The properties of a type-0 cipher are intended to guarantee that no information about the key or message used to create some piece of ciphertext can be discovered by only looking at the ciphertext. In our adversary model, information about the key or message in such ciphertexts may be obtained when either one is public (as opposed to in the typical adversary model for security protocol analysis where it would be true whenever the key was confidential). This follows from our assumption that all public values may become known to the adversary, therefore providing a means to determine when two encrypted public messages are different. Additionally, the adversary in our model may be able to determine when two ciphertexts containing public messages and created with different confidential keys are different. This is because API functions may exist which decrypt such messages. The ways in which an adversary may tell apart two ciphertexts are discussed further in §3.4.

Real-world ciphers, such as AES and DES (in CBC mode), are both which-key concealing and repetition concealing. Consequently, we need only ensure that an API implementation which uses such a cipher is message-length concealing when encrypting a confidential message with a confidential key. Provided the implementation never encrypts arbitrary length confidential messages, we can add padding to messages prior to encryption and our cipher will be message-length concealing.[2] The second property holds when encrypted messages have a known structure, such as the concatenation of the message and a hash value or checksum.

3.2.1 Confounders

Our analysis approach will use the symbolic model, as opposed to the computational model. However, since we assume that the encryption cipher is probabilistic, we need to capture this feature. Abadi and Rogaway showed that this can be achieved by using *confounders* [Abadi and Rogaway, 2000]. Confounders are random values that are freshly generated every time an encryption operation is carried out:

$$\begin{array}{ll} a := \mathrm{senc}(k_1, v_1); & a = \mathrm{ctxt}(n_1, k_1, v_1) \\ b := \mathrm{senc}(k_1, v_1); & b = \mathrm{ctxt}(n_2, k_1, v_1) \\ c := \mathrm{senc}(k_2, v_2); & c = \mathrm{ctxt}(n_3, k_2, v_2) \\ d := a; & d = \mathrm{ctxt}(n_1, k_1, v_1) \end{array}$$

Evaluating the code snippet on the left will result in the locations a through d containing the ciphertexts shown on the right. The contents of locations a and d are equal, and the contents of locations b and c are different to the contents of all other locations. Note that, when a piece of ciphertext is decrypted, the confounder is ignored.

This example shows that, given any pair of non-equal ciphertexts, we cannot say anything about whether or not the keys and messages are the same — the exact same property possessed by type-0 ciphers.

What may not be immediately apparent however, is that we must make a certain assumption about values in our system. This assumption is not unique to our system or analysis method, rather it arises from our use of the symbolic model. The symbolic model assumes that two values are equal if and only if they have the same structure. For non-compound values like keys, this means that every key identifier (e.g., k_1, k_2, etc.) represents a different bit-string. For ciphertexts, this means that two non-equal

[2] As security APIs typically protect sensitive cryptographic keys, or fixed-structure data items such as passwords, it is reasonable to assume this property of the systems being analysed.

ciphertexts always correspond to different bit-strings. In reality though, it is possible for two different key-message pairs to produce the same ciphertext. However, for two meaningful messages, it is a highly unlikely possibility and therefore is not considered a limitation when using the symbolic model.

3.3 Security Properties

In this section, we discuss the two main security properties that our system is designed to enforce: *confidentiality* and *non-interference*. Confidentiality is concerned with *direct* information flows, and non-interference is concerned with *indirect* information flows (although non-interference must also consider direct flows). Implicit flows arise from side-channels such as program control flow, execution time, resource consumption, etc..

3.3.1 Confidentiality

Confidentiality is the most intuitive information-flow property, and can be stated informally as follows:

> *A data item with an associated security level of l should never become known to principals whose clearance level is less than l*

Bell and LaPadula showed that it was possible to enforce confidentiality in a multi-level secure system by limiting just the read and write operations in that system [Bell and LaPadula, 1973]. For example, a *high* program should never write *high* data into a *low* location, and a *low* program should never be able to read the contents of a *high* location. We enforce these restrictions in our system through the typing rules for assignment and dereferencing.

3.3.2 Non-Interference

Non-interference is a more subtle information-flow property that covers both explicit and implicit flows, which can be stated informally as follows:

> *For a given security level l, changes to data items whose associated security level is greater than or equal to l should have no observable effect upon data items whose associated security level is less than l*

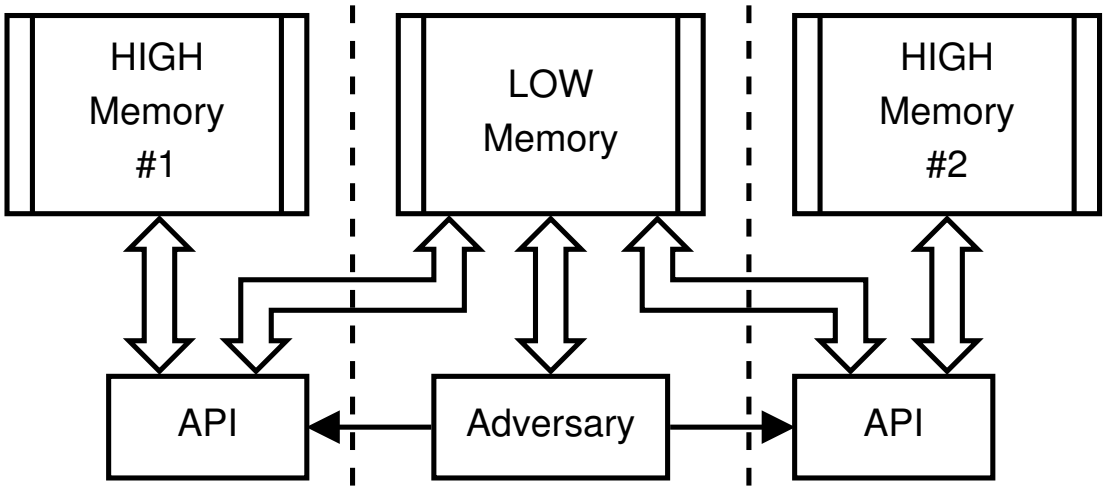

Figure 3.2: The Adversary Model for Indistinguishability

The following example shows an insecure information flow, arising from program control flow, which breaks the non-interference property:

$$\begin{aligned}&\text{if } (h == 1) \text{ then}\\ &\quad l := 1\\ &\text{else}\\ &\quad l := 0\end{aligned}$$

In this example, the *low* location l will contain 1 whenever the *high* value h equals 1. Consequently, there is an indirect information flow from h to l. This example breaks the non-interference property because there exists a pair of values for h which result in the contents of l being different, e.g., 0 and 1.

To prove that a system is non-interferent, we simulate two related executions of that system. This means we require an adversary model which takes into account two instances of the API, as shown in Fig. 3.2. The adversary interacts with the API in one of two configurations, and an API is said to be non-interferent when the adversary is unable to determine which of the two instances he is interacting with. The two instances of the API differ in the *high* values that are present in the system; all *low* values are equal. For example, each API may contain different confidential keys, and the API is non-interferent when this difference is transparent to the adversary. If this difference is visible to the adversary then this information may enable him to discover the cryptographic value of those confidential keys.

To prove that an API is non-interferent in K, we execute a program against each instance of the API and show that the adversary is unable to tell apart the two results

and final *low* memories. The comparison relation that the adversary uses to tell apart the two results is discussed in §3.4.

Indirect information flows like the one above can be prevented by restricting the assignments that may be made within the branches of a conditional test. For example, if the test involves *high* values, the branches may only modify the contents of *high* locations. In our system, we must also restrict function return values, since they may depend upon *high* data items.

3.4 Indistinguishability

To determine whether or not a particular program is non-interferent, it is necessary to compare *two* executions of that program — where data items in the *high* part of the system differ in each one. If the adversary can tell the two results apart, then there has been an insecure information flow from the *high* part of the system to the *low* part.

We now describe and justify the comparison relation used by the adversary. The formal relation is given in Definition 4.8 (Indistinguishability of Values); the relations outlined below show how this formal relation was reached. In the sections which follow, we write $v_1 \sim_i v_2$ to denote the comparison of values v_1 and v_2, where i is different for each subsequent version of the relation. In all cases, the adversary is unable to tell apart the two values when the relation holds.

3.4.1 Equality

The simplest check that the adversary can do is just to look at the two results and see if they are equal or not. This corresponds to structural equality in our abstract model of the system, and bit-wise equality in reality. This gives us the following comparison relation:

$$v_1 \sim_1 v_2 \text{ iff } v_1 = v_2$$

However, this is too simple a relation to use as it does not sufficiently deal with the comparison of ciphertext results.

3.4.2 Offline Decryption

In our system, where program results may be ciphertext, it is unrealistic to have an adversary who just does basic bit-string comparisons. Instead, we want to consider an

adversary who is able to decrypt any ciphertexts that he knows the correct key for.

To demonstrate how the comparison relation should deal with ciphertexts, let us consider the following examples, where $\mathcal{K}_{adv}$ is the set of keys known to the adversary:

- ctxt(n_1, k_1, v_1) and ctxt(n_2, k_2, v_2) where $k_1, k_2 \in \mathcal{K}_{adv}$

  Here, the adversary can correctly decrypt both ciphertexts, thereby obtaining v_1 and v_2. There are two ways in which the adversary would be able to tell the ciphertexts apart. The first is if he can tell apart v_1 and v_2; the second is if k_1 and k_2 are non-equal. In the first case, the adversary knows that the messages are different, and in the second case, he knows that a different key has been used to create each ciphertext.

- ctxt(n_1, k_1, v_1) and ctxt(n_2, k_2, v_2) where $k_1, k_2 \notin \mathcal{K}_{adv}$

  Here, the adversary is unable to decrypt either ciphertext. Consequently, since we assume a type-0 cipher, the adversary cannot obtain any information about the keys or messages in either one. As such, the ciphertexts can be viewed as random bit-strings. These bit-strings will be equal if the confounders are equal, and will be non-equal if the confounders are different.[3] In both cases however, since no sensitive information can be obtained by the adversary, we want the comparison relation to hold (i.e., the adversary cannot tell the ciphertexts apart).

- ctxt(n_1, k_1, v_1) and ctxt(n_2, k_2, v_2) where $k_1 \in \mathcal{K}_{adv}$ and $k_2 \notin \mathcal{K}_{adv}$

  Here, the adversary is able to correctly decrypt one ciphertext, but not the other. As such, he knows that a different key has been used to create each one, and therefore that they are different. The ability to determine that one ciphertext has been correctly decrypted but not the other follows from our requirement that we can determine whether or not the correct key has been used in a decryption operation (see §3.2).

These examples show when two potentially non-equal pieces of ciphertext can or cannot be told apart. As a direct result of the second example, our comparison relation equates more pairs of results than those which are bit-wise equal. This follows from our desire to capture the information-hiding capabilities of probabilistic encryption ciphers.

[3] The probabilistic nature of the encryption cipher makes it statistically unlikely that two different encryption operations will produce equal ciphertexts.

Our comparison relation is now defined as follows:

$$\begin{aligned} v_1 &\sim_2 v_2 \quad \text{when} \quad v_1 = v_2 \\ \text{ctxt}(n_1, k_1, v_1) &\sim_2 \text{ctxt}(n_2, k_2, v_2) \quad \text{when} \quad k_1, k_2 \notin \mathcal{K}_{adv} \\ \text{ctxt}(n_1, k_1, v_1) &\sim_2 \text{ctxt}(n_2, k_2, v_2) \quad \text{when} \quad k_1, k_2 \in \mathcal{K}_{adv}, k_1 = k_2 \text{ and } v_1 \sim_2 v_2 \end{aligned}$$

3.4.3 API Functions

The relation that we will use when comparing the outputs from two executions of a particular program will be a formal version of the one shown in §3.4.2. However, an alternative version of the relation is necessary when we allow the adversary to make calls to arbitrary API functions. The differences between the previous relation and the one given below stem from the fact that, in general, API functions will be able to carry out successful decryption operations using a larger set of keys (i.e., all the keys known to the adversary, plus some or all of those from the *high* part of the system). To demonstrate this further, consider the following examples, where $\mathcal{K}_{api}$ is the set of keys which are usable in decryption operations by API code, and $\mathcal{L}$ is the set of all data items in the *low* part of the system:

- $\text{ctxt}(n_1, k_1, v_1)$ and $\text{ctxt}(n_2, k_2, v_2)$ where $k_1, k_2 \in (\mathcal{K}_{api} \setminus \mathcal{K}_{adv})$ and $v_1, v_2 \in \mathcal{L}$

  Here, the adversary is unable to decrypt the two pieces of ciphertext himself. However, it is possible that he can use a sequence of API calls to obtain v_1 and/or v_2. In the general case, we cannot know what functions a particular API will provide. As such, we must assume the worst case where the adversary is able to use one or more API functions to obtain both v_1 and v_2.

  The adversary can therefore distinguish the two ciphertexts when the process required to obtain v_1 is different to that which is required to obtain v_2, or when v_1 and v_2 are themselves distinguishable.

- $\text{ctxt}(n_1, k_1, v_1)$ and $\text{ctxt}(n_2, k_2, v_2)$ where $k_1, k_2 \in (\mathcal{K}_{api} \setminus \mathcal{K}_{adv})$ and $v_1, v_2 \notin \mathcal{L}$

  Here, the adversary is unable to decrypt the two pieces of ciphertext himself and, because the messages are not *low*, a secure API will not allow the adversary to obtain them.[4] Consequently, the adversary will be unable to tell apart these two ciphertexts.

[4] We can assume that the API is secure in this relation because it will not be used to compare outputs, only to restrict inputs. That is, if the API is not actually secure, then a program will exist that produces output which can be distinguished based on the relation given in §3.4.2. See the discussion which follows for more details.

The ability for the adversary to use API functions to tell apart data items is equivalent to saying that the adversary can use any client program to try and tell them apart. In other words, the adversary can use a single client program, taking a single input, and execute it twice; once with one data item and once with the other. Since there are no restrictions on the length of such programs, the final comparison of the two program outputs can be done using the relation described in §3.4.2.

Additionally, this means that program inputs will only be considered the same if they satisfy the relation given in this section. This relation is as follows, where the sets are the same as previously defined:

$$\begin{aligned} v_1 &\sim_3 v_2 \quad \text{when} \quad v_1 = v_2 \\ \text{ctxt}(n_1, k_1, v_1) &\sim_3 \text{ctxt}(n_2, k_2, v_2) \quad \text{when} \quad k_1, k_2 \notin \mathcal{K}_{adv} \text{ and } v_1, v_2 \notin \mathcal{L} \\ \text{ctxt}(n_1, k_1, v_1) &\sim_3 \text{ctxt}(n_2, k_2, v_2) \quad \text{when} \quad v_1, v_2 \in \mathcal{L}, k_1 = k_2 \text{ and } v_1 \sim_3 v_2 \end{aligned}$$

It is straightforward to show that two results satisfying this comparison relation also satisfy the relation given in §3.4.2:

- In the first case, the result is immediate since the same case exists in the previous relation.
- In the second case, we have $k_1, k_2 \notin \mathcal{K}_{adv}$ and this is sufficient to satisfy the second case of the previous relation.
- In the third case, we have $k_1 = k_2$ and $v_1 \sim_3 v_2$. Since the symbolic model requires that data items are distinct, it must be the case that k_1 and k_2 are either both in $\mathcal{K}_{adv}$ or are both not in $\mathcal{K}_{adv}$. In the former instance, the third case of the previous relation is satisfied, and in the latter instance, the second case of the previous relation is satisfied.

3.4.4 Observation Levels

The comparison relations given in the previous sections involve the set of keys known to the adversary, $\mathcal{K}_{adv}$. We can over-approximate this set with $\mathcal{K}_{\mathcal{L}}$, the set of all keys in the *low* part of the system. Recall that a key is in $\mathcal{K}_{\mathcal{L}}$ if the security label associated with it is dominated in the security lattice by the adversary's observation level (i.e., the adversary is permitted to know that key). Consequently, we can check whether or not a key is in the set $\mathcal{K}_{\mathcal{L}}$ by looking at the security level associated with it.

If l is the observation level of the adversary, then the security level of every data item in $\mathcal{K}_L$ is less than or equal to l. We can therefore state the comparison relation from §3.4.3 as follows:

$$
\begin{aligned}
v_1 \sim_3 v_2 \quad &\text{when} \quad v_1 = v_2 \\
\mathrm{ctxt}(n_1, k_1, v_1) \sim_3 \mathrm{ctxt}(n_2, k_2, v_2) \quad &\text{when} \quad lvl(k_1) \not\leq l,\ lvl(k_2) \not\leq l,\ lvl(v_1) \not\leq l \text{ and} \\
&\qquad\quad lvl(v_2) \not\leq l \\
\mathrm{ctxt}(n_1, k_1, v_1) \sim_3 \mathrm{ctxt}(n_2, k_2, v_2) \quad &\text{when} \quad k_1 = k_2,\ lvl(v_1) \leq l,\ lvl(v_2) \leq l \text{ and } v_1 \sim_3 v_2
\end{aligned}
$$

Similarly, the comparison relation from §3.4.2 can be stated as follows:

$$
\begin{aligned}
v_1 \sim_2 v_2 \quad &\text{when} \quad v_1 = v_2 \\
\mathrm{ctxt}(n_1, k_1, v_1) \sim_2 \mathrm{ctxt}(n_2, k_2, v_2) \quad &\text{when} \quad lvl(k_1) \not\leq l \text{ and } lvl(k_2) \not\leq l \\
\mathrm{ctxt}(n_1, k_1, v_1) \sim_2 \mathrm{ctxt}(n_2, k_2, v_2) \quad &\text{when} \quad lvl(k_1) \leq l,\ lvl(k_2) \leq l,\ k_1 = k_2 \text{ and } v_1 \sim_2 v_2
\end{aligned}
$$

Chapter 4

Formal System

In this chapter we present the formal model of our type system. Section 4.1 gives the abstract syntax, Section 4.2 describes the evaluation relation, Section 4.3 outlines the major security properties which it is intended to enforce, and Section 4.4 defines the typing rules and associated lemmas. The proofs of the main security properties are presented in Section 4.5, with all other proofs given in Appendix A.

4.1 Abstract Syntax

We define the following (countably infinite) disjoint sets of identifiers, with elements of each denoted by the specified meta-variable:

$l \in \mathcal{L}$	*Security labels*
$c \in \mathcal{C}$	*Constants*
$k \in \mathcal{K}$	*Keys*
$a \in \mathcal{A}$	*Memory addresses (locations)*
$f \in \mathcal{F}$	*Function identifiers*
$x \in \mathcal{X}$	*Variables*
$n \in \mathcal{N}$	*Nonces*

The set $\mathcal{L}$ defines the possible security labels which may be associated with terms in the system (e.g., $\{$H, L$\}$ or $\{$TOP SECRET, SECRET, CONFIDENTIAL, RESTRICTED, UNCLASSIFIED$\}$). We obtain a *security level lattice* $(\mathcal{L}, \leq)$ by giving a partial order on $\mathcal{L}$ such that there exists a least element, denoted $\bot$, and a greatest element, denoted $\top$. The ordering relation $\leq$ must be transitive and reflexive, as we rely upon these properties in many of the proofs presented in this thesis. The lattice operations, $\curlyvee$ and $\curlywedge$, are assumed to follow the standard semantics for least upper bound and greatest

lower bound, respectively. From this point onwards, we assume that there exists an unspecified but fixed security level lattice which has the required properties. Constants and keys both correspond to arbitrary but fixed bit-strings, with constants representing regular data items and keys able to be used for encryption and decryption purposes. Memory addresses correspond to arbitrary locations in memory, of no particular size but large enough to hold each value stored in them. We consider memory addresses to be unrelated to each other, and therefore do not model low-level operations such as pointer arithmetic. This view follows from our intention that K focuses on preventing information leaks which arise from the logical execution path of a particular program with respect to a given API. By abstracting away certain properties of the underlying physical system, we limit the complexity of our system to just those areas where it is necessary to study the problem of interest. Nonces represent confounders and allow us to represent non-deterministic encryption. Confounders were first introduced in [Abadi and Rogaway, 2000], and the motivation behind their use was discussed in §3.2.1.

We consider systems (i.e., programs) comprising an API definition and a single executable block of code, where the bodies of API functions and the executable code are drawn from the same set of expressions, as defined by the following term grammar:

$$
\begin{array}{lll}
v & ::= & c \mid a \mid k \mid \mathrm{ctxt}(n, k, v) \mid \mathrm{TRUE} \mid \mathrm{FALSE} \qquad \textit{Values} \\
e & ::= & v \mid x \mid \mathrm{senc}(e, e) \mid \mathrm{try}\ \mathrm{sdec}(x, x) = x\ \mathrm{in}\ e\ \mathrm{else}\ e \\
 & & \quad \mid \mathrm{if}\ x\ \mathrm{then}\ e\ \mathrm{else}\ e \mid e == e \mid e := e \mid *e \mid f(\overrightarrow{e}) \qquad \textit{Expressions} \\
d & ::= & d\,;\, l\ c \mid d\,;\, l\ k \mid d\,;\, l\ a \mid d\,;\, l\ f(\overrightarrow{x})\,\{e\} \mid \varepsilon \qquad \textit{API definition} \\
p & ::= & (d, e) \qquad \textit{Program}
\end{array}
$$

An API definition is a possibly empty sequence of constant, key, memory address and function definitions. Constant and key definitions comprise an identifier preceded by the security label which that term should be associated with. Definitions of memory addresses also comprise an identifier preceded by a security label, but here the security label denotes the maximum level of values which may be stored in the corresponding location; the security level associated with the identifier is $\bot$. Function definitions comprise the function identifier, a possibly empty list of parameter variables and an expression corresponding to the function body. Functions definitions are also preceded by a security label which denotes a lower bound on the security level of the return value and of any modified locations. This lower bound marks the clearance level below which the function call should have no observable effect.

4.1.1 Definitions

We now present some standard definitions that we make use of in this chapter.

Definition 4.1 (Substitution)**.**

The substitution of a value v for a variable x within an expression e, written [v/x]e is defined inductively by the following rules:

$$
\begin{aligned}
&[v/x]v' = v' \\
&[v/x]x = v \\
&[v/x]x' = x' \qquad (x' \neq x) \\
&[v/x]\mathit{let}\ x' = e_1\ \mathit{in}\ e_2 = \mathit{let}\ x' = [v/x]e_1\ \mathit{in}\ [v/x]e_2 \qquad (x' \neq x) \\
&[v/x]\mathit{senc}(e_k, e_m) = \mathit{senc}([v/x]e_k, [v/x]e_m) \\
&[v/x]\mathit{try}\ \mathit{sdec}(e_k, e_c) = x'\ \mathit{in}\ e_1\ \mathit{else}\ e_2 \\
&\qquad\qquad = \mathit{try}\ \mathit{sdec}([v/x]e_k, [v/x]e_c) = x'\ \mathit{in}\ [v/x]e_1\ \mathit{else}\ [v/x]e_2 \qquad (x' \neq x) \\
&[v/x]\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 = \mathit{if}\ [v/x]e_1\ \mathit{then}\ [v/x]e_2\ \mathit{else}\ [v/x]e_3 \\
&[v/x]{*}e = {*}[v/x]e \\
&[v/x]e_1 := e_2 = [v/x]e_1 := [v/x]e_2 \\
&[v/x]f(e_1, \ldots, e_n) = f([v/x]e_1, \ldots, [v/x]e_n)
\end{aligned}
$$

Values are unchanged by substitution since, by definition, they cannot contain any variables. Other expressions have the substitution applied to their sub-expressions. Note that, for function calls, no substitution is done in the body of the function. The substitution of variables within function bodies is restricted to those variables defined in the function parameter list, rather than being ruled out altogether.

The $x \neq x'$ restriction in the 'let' and 'try sdec' cases is easily ensured by the following convention, which allows us to rename any bound variable to a free identifier, provided that all occurrences of that variable are similarly renamed:

Convention 4.2 (Alpha Renaming)**.**

Terms that differ only in the names of bound variables are interchangeable in all contexts.

For example, to substitute v' for x in the expression let $x = v$ in x, we first rename the variable in the let expression to x' then proceed with $[v'/x](\text{let } x' = v \text{ in } x')$, yielding the result let $x' = v$ in x'.

Definition 4.3 (Free Variables)**.**

The set of free variables *of an expression e, written FV(e), is defined as follows:*

$$
\begin{array}{lll}
FV(v) = \emptyset & & FV(\text{let } x = e_1 \text{ in } e_2) = FV(e_1) \cup (FV(e_2) \setminus \{x\}) \\
FV(x) = \{x\} & & FV(f(e_1, \ldots, e_n)) = \bigcup_{i=1}^{n} FV(e_i) \\
 & & FV(senc(e_k, e_m)) = FV(e_k) \cup FV(e_m) \\
\multicolumn{3}{l}{FV(\text{try } sdec(e_k, e_c) = x \text{ in } e_1 \text{ else } e_2) = FV(e_k) \cup FV(e_c) \cup (FV(e_1) \cup FV(e_2) \setminus \{x\})} \\
\multicolumn{3}{l}{FV(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) = \text{if } FV(e_1) \text{ then } FV(e_2) \text{ else } FV(e_3)} \\
\multicolumn{2}{l}{FV(e_1 := e_2) = FV(e_1) \cup FV(e_2)} & FV(*e) = FV(e)
\end{array}
$$

Definition 4.4 (Closed Expression)**.**

An expression e is defined to be closed *when FV(e) = ∅.*

4.2 Evaluation Relation

Expressions (and therefore client programs) are evaluated with respect to an *evaluation environment* Δ and *evaluation context* Π:

$$\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta', v \rangle \qquad \textit{Evaluation relation}$$

The left and right side of an evaluation relation will be referred to as an *execution state*, or simply a *state*. The evaluation context is a finite set of variable to value mappings, and the evaluation environment is a tuple comprising a finite set of value to security level mappings, a finite set of function definitions, a potentially infinite set of confounders, and a *store*:

$$
\begin{array}{lll}
\Pi & ::= & \{x_i \mapsto v_i \mid i \in 1..n\} \qquad \textit{Evaluation context} \\
\Delta & ::= & (\mathbb{L}, \mathbb{F}, \mathbb{N}, \phi) \qquad \textit{Evaluation environment} \\
(\Pi, \Delta) & & \textit{Evaluation domain}
\end{array}
$$

The four components comprising the evaluation environment have the following forms:

$$
\begin{array}{lll}
\mathbb{L} & ::= & \{v : l \mid v \in \{c, k, a\}\} \qquad \textit{Security levels mapping} \\
\mathbb{F} & ::= & f(x_1, \ldots, x_n) = e : l, \mathbb{F} \mid \varepsilon \qquad \textit{Function definitions} \\
\mathbb{N} & ::= & n, \mathbb{N} \mid \varepsilon \qquad \textit{Confounders} \\
\phi & ::= & (a \mapsto v), \phi \mid \varepsilon \qquad \textit{Store}
\end{array}
$$

The set of security level mappings in the initial evaluation environment is generated from the definition of the API that the expression is to be evaluated against. This set only contains mappings for constants, keys and memory locations because the security

$$\frac{0 = \textit{default value for } x}{(\varepsilon, e) \rightsquigarrow (\{x \mapsto 0 \mid x \in FV(e)\}, \emptyset)}\ \text{ED-Empty}$$

$$\frac{(d, e) \rightsquigarrow (\Pi, \Delta) \qquad c \notin \Delta}{(d\,;\, c : l, e) \rightsquigarrow (\Pi, \Delta \cup c : l)}\ \text{ED-Cnst}$$

$$\frac{(d, e) \rightsquigarrow (\Pi, \Delta) \qquad k \notin \Delta}{(d\,;\, k : l, e) \rightsquigarrow (\Pi, \Delta \cup k : l)}\ \text{ED-Key}$$

$$\frac{(d, e) \rightsquigarrow (\Pi, \Delta) \qquad a \notin \Delta}{(d\,;\, a : l, e) \rightsquigarrow (\Pi, \Delta \cup a : l)}\ \text{ED-Loc}$$

$$\frac{(d, e) \rightsquigarrow (\Pi, \Delta) \qquad f \notin \Delta}{(d\,;\, l\ f(\overrightarrow{x})\ \{e'\}, e) \rightsquigarrow (\Pi, \Delta \cup f(\overrightarrow{x}) = e' : l)}\ \text{ED-Func}$$

Figure 4.1: Construction rules for evaluation domains

levels associated with other values are either fixed, or are based on the security levels of their sub-expressions. The function which maps a value to its associated security level is given in Definition 4.5. The set of function definitions is also generated from the API definition, and it remains fixed throughout the evaluation of an expression. The set of confounders is initially empty, and is added to whenever an encryption operation occurs. This set is used to ensure that each new confounder is fresh. The store contains exactly those locations for which there are security level mappings in $\mathbb{L}$.

For brevity, the following shorthand notations are used throughout this chapter, where $\Delta = (\mathbb{L}, \mathbb{F}, \mathbb{N}, \phi)$:

$$\begin{array}{rcl@{\qquad}rcl}
c \notin \Delta & \equiv & c \notin \mathsf{dom}(\mathbb{L}) & \Delta \cup c : l & \equiv & ((\mathbb{L}, c : l), \mathbb{F}, \mathbb{N}, \phi) \\
k \notin \Delta & \equiv & k \notin \mathsf{dom}(\mathbb{L}) & \Delta \cup k : l & \equiv & ((\mathbb{L}, k : l), \mathbb{F}, \mathbb{N}, \phi) \\
a \notin \Delta & \equiv & a \notin \mathsf{dom}(\mathbb{L}) & \Delta \cup a : l & \equiv & ((\mathbb{L}, a : l), \mathbb{F}, \mathbb{N}, \phi) \\
n \notin \Delta & \equiv & n \notin \mathsf{dom}(\mathbb{N}) & \Delta \cup n & \equiv & (\mathbb{L}, \mathbb{F}, (\mathbb{N}, n), \phi) \\
f \notin \Delta & \equiv & f \notin \mathsf{dom}(\mathbb{F}) & \Delta \cup f(\overrightarrow{x}) = e' : l & \equiv & (\mathbb{L}, (\mathbb{F}, f(\overrightarrow{x}) = e' : l), \mathbb{N}, \phi) \\
\Delta(a) & \equiv & \phi(a) & \Delta[a \mapsto v] & \equiv & (\mathbb{L}, \mathbb{F}, \mathbb{N}, \phi[a \mapsto v])
\end{array}$$

Figure 4.1 presents the rules for generating an initial evaluation domain from a program definition, while Fig. 4.2 shows a basic API definition and program expression, along with the resulting initial evaluation context and initial evaluation environment.

```
HIGH km      // secret master key
HIGH KEY     // location to store encryption key
HIGH k1      // arbitrary secret encryption key
LOW msg      // arbitrary public message

// Updates the stored encryption key. The supplied cipher text is decrypted with the
// master key and, if successful, the result is stored as the encryption key. Returns
// TRUE on success and FALSE otherwise.
LOW
set_key(wKey) {
  try sdec(km, wKey) = k in
    let x = (KEY := k) in TRUE
  else FALSE
}

// Encrypts the given message with the currently stored encryption key.
LOW
encrypt_msg(msg) {
  senc(*KEY, msg)
}


// Sets the encryption key to k1 and then encrypts msg.
let x = set_key(ctxt(n1, km, k1)) in encrypt_msg(msg)
```

$$
\begin{aligned}
\Pi &= \varepsilon \\
\mathbb{L} &= \mathsf{km} : \text{HIGH},\ \mathsf{k1} : \text{HIGH},\ \mathsf{msg} : \text{LOW},\ \text{KEY} : \text{HIGH} \\
\mathbb{F} &= \text{set\_key}(wKey) = \\
&\qquad (\text{try sdec}(wKey, k) = \mathsf{km} \text{ in } (\text{let } x = (\text{KEY} := k) \text{ in TRUE}) \text{ else FALSE}) : \text{LOW}, \\
&\qquad \text{encrypt\_msg}(msg) = \\
&\qquad\quad \text{senc}(*\text{KEY}, msg) : \text{LOW} \\
\mathbb{N} &= \mathsf{n1} \\
\phi &= (\text{KEY} \mapsto 0)
\end{aligned}
$$

Figure 4.2: Simple API and program definition, with resulting initial evaluation context and environment.

Read and write operations on the store are defined as follows:

$$\begin{array}{rcll} \phi(a) & \equiv & v & \textit{where} \quad (a \mapsto v) \in \phi \qquad \textit{Read} \\ \phi[a \mapsto v] & \equiv & \phi' & \textit{where} \quad \mathsf{dom}(\phi') = \mathsf{dom}(\phi),\ (a \mapsto v) \in \phi' \textit{ and} \\ & & & \quad \forall a' \in \mathsf{dom}(\phi')\ ((a' \neq a \wedge (a' \mapsto v') \in \phi) \rightarrow (a' \mapsto v') \in \phi') \qquad \textit{Write} \end{array}$$

We assume that locations are never read from before they have been assigned to for the first time. Since we do not allow new locations to be created at execution time, this can be guaranteed by initialising the store to contain default values for each location.

We define the function $lvl_\Delta(\cdot)$ which maps values to security levels, in accordance with a set of security level mappings, as follows:

Definition 4.5 (Security Level Function)**.**

The security level of a value, with respect to a set of security level mappings $\mathbb{L}$ *from some evaluation environment* Δ*, is given by the function* $lvl_\Delta(\cdot)$*, corresponding to the least relation defined inductively by the following rules:*

$$\begin{array}{rcll} lvl_\Delta(c) & = & l & \textit{iff } c : l \in \mathbb{L} \\ lvl_\Delta(k) & = & l & \textit{iff } k : l \in \mathbb{L} \\ lvl_\Delta(a) & = & \bot & \\ lvl_\Delta(ctxt(n, v_k, v_m)) & = & \begin{cases} \bot & \textit{if } lvl_\Delta(v_m) \leq lvl_\Delta(v_k) \\ lvl_\Delta(v_m) & \textit{otherwise} \end{cases} & \\ lvl_\Delta(\text{TRUE}) & = & \bot & \\ lvl_\Delta(\text{FALSE}) & = & \bot & \end{array}$$

The security level of a constant or key is taken from the evaluation environment, while memory addresses and Boolean values are defined to be public. Boolean values are public because the two possible values are common knowledge, however this does not preclude the ability to have a confidential Boolean variable. Memory addresses are public because knowledge of a memory address does not mean that the contents of the corresponding location can be read. This distinction lets us use memory addresses as *key handles* when modelling Cachin and Chandran's API in Chapter 5. Ciphertext is defined to be public when the security level of the key dominates that of the message, as only those people with knowledge of the key are able to recover the message, thus the ciphertext can be declassified. When the security level of the key is lower than that of the message, the ciphertext cannot be declassified, since someone who is not permitted to know the message may have the key and be able to decrypt the ciphertext.

A store is said to be *valid* when the security level of the values stored in each location are no greater than the maximum level which the API definition states may be stored in that location:

Definition 4.6 (Valid Store)**.**

A store ϕ *is said to be* valid *with respect to some set of security level mappings* $\mathbb{L}$*, written* $\mathbb{L} \vdash \phi$*, iff* $(a : l \in \mathbb{L}) \rightarrow (a \in \mathsf{dom}(\phi) \wedge lvl_{\Delta}(\phi(a)) \leq l)$*.*

Intuitively, a store is valid provided that when a user is permitted to read some memory location, they are permitted to know the contents of that location.

We are now ready to present the evaluation rules for K, and these are shown in Fig. 4.3. Informally, the evaluation rules have the following semantics:

E-VAL Has no effect. This rule exists to help simplify the definition of certain Lemmas and Theorems by avoiding the need to distinguish between the expression being a value or not.

E-VAR Returns the value that the variable is mapped to in the evaluation context.

E-LET Adds a new variable to the evaluation context, mapped to the value that the first expression evaluates to. The second expression is then evaluated against this updated context.

E-SENC This rule corresponds to a symmetric encryption operation. The key and message expressions are evaluated, and then ciphertext is returned which includes a fresh confounder. As noted in §3.1.1, the confounder is used to model the probabilistic nature of the encryption cipher (i.e., different encryption operations that use the same key and message will produce ciphertexts with different confounders and thus they will be non-equal).

E-SDEC1 This rule corresponds to a successful decryption operation. The key and ciphertext expressions are evaluated and the ciphertext has been encrypted with the same key. The plaintext is assigned to the specified, new variable with the result of evaluating the first branch expression under the extended evaluation context is then returned.

E-SDEC2 This rule corresponds to an unsuccessful decryption operation. The key and ciphertext expressions are evaluated and either the ciphertext has not been encrypted with the same key, or it is not even ciphertext. The result of evaluating the second branch expression is then returned.

$$\frac{}{\Pi \vdash \langle \Delta, v \rangle \Downarrow \langle \Delta, v \rangle}\ \text{E-VAL} \qquad \frac{(x \mapsto v) \in \Pi}{\Pi \vdash \langle \Delta, x \rangle \Downarrow \langle \Delta, v \rangle}\ \text{E-VAR}$$

$$\frac{\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta', v_1 \rangle \qquad x \notin \mathsf{dom}(\Pi) \qquad \Pi \cup (x \mapsto v_1) \vdash \langle \Delta', e_2 \rangle \Downarrow \langle \Delta'', v_2 \rangle}{\Pi \vdash \langle \Delta, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow \langle \Delta'', v_2 \rangle}\ \text{E-LET}$$

$$\frac{\Pi \vdash \langle \Delta, e_k \rangle \Downarrow \langle \Delta', k \rangle \qquad \Pi \vdash \langle \Delta', e_m \rangle \Downarrow \langle \Delta'', v \rangle \qquad n \notin \Delta''}{\Pi \vdash \langle \Delta, \text{senc}(e_k, e_m) \rangle \Downarrow \langle (\Delta'' \cup n), \text{ctxt}(n, k, v) \rangle}\ \text{E-SENC}$$

$$\frac{\begin{array}{c}\Pi \vdash \langle \Delta, e_k \rangle \Downarrow \langle \Delta', k \rangle \qquad \Pi \vdash \langle \Delta', e_c \rangle \Downarrow \langle \Delta'', \text{ctxt}(n, k, v_m) \rangle \\ x \notin \mathsf{dom}(\Pi) \qquad \Pi \cup (x \mapsto v_m) \vdash \langle \Delta'', e_1 \rangle \Downarrow \langle \Delta''', v \rangle\end{array}}{\Pi \vdash \langle \Delta, \text{try sdec}(e_k, e_c) = x \text{ in } e_1 \text{ else } e_2 \rangle \Downarrow \langle \Delta''', v \rangle}\ \text{E-SDEC1}$$

$$\frac{\begin{array}{c}\Pi \vdash \langle \Delta, e_k \rangle \Downarrow \langle \Delta', k \rangle \qquad \Pi \vdash \langle \Delta', e_c \rangle \Downarrow \langle \Delta'', v_c \rangle \\ v_c \neq \text{ctxt}(n, k, v_m) \qquad \Pi \vdash \langle \Delta'', e_2 \rangle \Downarrow \langle \Delta''', v \rangle\end{array}}{\Pi \vdash \langle \Delta, \text{try sdec}(e_k, e_c) = x \text{ in } e_1 \text{ else } e_2 \rangle \Downarrow \langle \Delta''', v \rangle}\ \text{E-SDEC2}$$

$$\frac{\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta', v \rangle \qquad \Pi \vdash \langle \Delta', e_2 \rangle \Downarrow \langle \Delta'', v \rangle}{\Pi \vdash \langle \Delta, e_1 == e_2 \rangle \Downarrow \langle \Delta'', \text{TRUE} \rangle}\ \text{E-EQ1}$$

$$\frac{\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta', v_1 \rangle \qquad \Pi \vdash \langle \Delta', e_2 \rangle \Downarrow \langle \Delta'', v_2 \rangle \qquad v_1 \neq v_2}{\Pi \vdash \langle \Delta, e_1 == e_2 \rangle \Downarrow \langle \Delta'', \text{FALSE} \rangle}\ \text{E-EQ2}$$

$$\frac{\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta', \text{TRUE} \rangle \qquad \Pi \vdash \langle \Delta', e_2 \rangle \Downarrow \langle \Delta'', v \rangle}{\Pi \vdash \langle \Delta, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rangle \Downarrow \langle \Delta'', v \rangle}\ \text{E-IF1}$$

$$\frac{\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta', \text{FALSE} \rangle \qquad \Pi \vdash \langle \Delta', e_3 \rangle \Downarrow \langle \Delta'', v \rangle}{\Pi \vdash \langle \Delta, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rangle \Downarrow \langle \Delta'', v \rangle}\ \text{E-IF2}$$

$$\frac{\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta', a \rangle \qquad \Pi \vdash \langle \Delta', e_2 \rangle \Downarrow \langle \Delta'', v \rangle}{\Pi \vdash \langle \Delta, e_1 := e_2 \rangle \Downarrow \langle \Delta''[a \mapsto v\ ], v \rangle}\ \text{E-ASGN}$$

$$\frac{\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta', a \rangle}{\Pi \vdash \langle \Delta, *e \rangle \Downarrow \langle \Delta', \Delta'(a) \rangle}\ \text{E-DREF}$$

$$\frac{\begin{array}{c}\forall i \in 1..n\ \ \Pi \vdash \langle \Delta_{i-1}, e_i \rangle \Downarrow \langle \Delta_i, v_i \rangle \qquad f(x_{i\ i \in 1..n}) = e_f \in \Delta_n \\ x'_i \notin \mathsf{dom}(\Pi) \qquad x'_i \notin FV(e_f) \qquad (\Pi \cup x'_i \mapsto v_i) \vdash \langle \Delta_n, [x'_i / x_i] e_f \rangle \Downarrow \langle \Delta', v \rangle\end{array}}{\Pi \vdash \langle \Delta_0, f(e_{i\ i \in 1..n}) \rangle \Downarrow \langle \Delta', v \rangle}\ \text{E-FNAPP}$$

Figure 4.3: Evaluation rules in K

E-EQ1 This rule corresponds to a successful equality test. The two expressions are evaluated and produce the same value, thus TRUE is returned.

E-EQ2 This rule corresponds to an unsuccessful equality test. The two expressions are evaluated and produce different values, thus FALSE is returned.

E-IF1 The guard expression is evaluated and produces TRUE, and the result of the evaluating the first branch expression is then returned.

E-IF2 The guard expression is evaluated and produces FALSE, and the result of the evaluating the first branch expression is then returned.

E-ASGN Updates the value stored in the location that the expression evaluates to.

E-DREF Returns the value stored in the location that the expression evaluates to.

E-FNAPP This rule corresponds to calling a pre-defined function. If present, the function arguments are evaluated, in left-to-right order, then the function argument variables in the function body expression are replaced with fresh variables (to avoid variable capture in case that the same function is called multiple times). This modified function body expression is then evaluated with respect to an extended evaluation context that includes the function argument variable mappings.

Observe that none of the evaluation rules include security level checks. This ensures that all of the security guarantees follow exclusively from the typing rules (see Figs. 4.4 and 4.5).

4.3 Security Properties

The two security properties which K is designed to enforce are *confidentiality* and *non-interference*.

4.3.1 Confidentiality

Definition 4.7 (Confidentiality-Preserving Expression)**.**

An expression e is confidentiality-preserving *with respect to an evaluation domain* (Π, Δ) *and security level l if whenever* $\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta', v \rangle$*, we have:*

- $lvl_{\Delta'}(v) \leq l$
- $(a : l' \in \Delta' \wedge a \mapsto v' \in \Delta') \rightarrow lvl_{\Delta'}(v') \leq l'$

This property states that, for some initial evaluation domain (Π, Δ) and observation level l, an expression is *confidentiality-preserving* when, after evaluation, the security level of the result is no greater than l and the security level of each value stored in a location is no greater than that which the location may store. In other words, no write-downs will occur as a result of evaluating the expression against the given initial evaluation domain.

4.3.2 Non-Interference

The non-interference property is defined formally in terms of two related program executions. Specifically, given an expression and two evaluations which differ only in the non-observable part of their initial state, the adversary should be unable to detect any differences between the results of the two evaluations. The result of an evaluation is the value which the expression evaluates to, as well as the observable part of the state at each step of the evaluation. We do not attempt to address differences such as evaluation time. However, before we present our definition of non-interference, we must first provide a few supporting definitions.

Indistinguishability of Values

Central to the non-interference property is the notion of what it means for two values to be indistinguishable. Definition 4.8 presents the formal relation that we will use, and is based on the informal relation described in Section 3.4.2. This definition is equivalent to the original version given in [McCullough, 1988] and is a weakening of equality that takes into account the information-hiding properties of probabilistic ciphers.

Definition 4.8 (Indistinguishability of Values)**.**

With respect to some security level mapping $\mathbb{L}$, two values, v and v', are defined to be indistinguishable *at the observation level l, written $\mathbb{L} \vdash v \approx_l v'$, if one or more of the following conditions hold:*

- $v = v'$
- $v = ctxt(n, k, v_m)$, $v' = ctxt(n', k', v'_m)$, $lvl_{\mathbb{L}}(k) \not\leq l$ *and* $lvl_{\mathbb{L}}(k') \not\leq l$
- $v = ctxt(n, k, v_m)$, $v' = ctxt(n', k', v'_m)$, $k = k'$ *and* $\mathbb{L} \vdash v_m \approx_l v'_m$

To aid the understanding of this definition, we now present some examples of both indistinguishable and distinguishable pairs of values, where the subscripts on constants and keys denote the associated security level:

1. $\mathbb{L} \vdash c_{\mathsf{H}} \not\approx_{\mathsf{L}} c'_{\mathsf{H}}$, where $c_{\mathsf{H}} \neq c'_{\mathsf{H}}$

   When the indistinguishability relation is applied to two non-ciphertext values, the security levels associated with those values are ignored. If c_{H} and c'_{H} were deemed to be indistinguishable at the observation level L, the relation would still hold when the observer could obtain two high security, but non-equal, values — a direct information leak which we want the relation to be able to detect.

2. $\mathbb{L} \vdash \mathrm{ctxt}(n, k_{\mathsf{H}}, c_{\mathsf{L}}) \approx_{\mathsf{L}} \mathrm{ctxt}(n', k_{\mathsf{H}}, c'_{\mathsf{L}})$, where $c_{\mathsf{L}} \neq c'_{\mathsf{L}}$

   Since the key used to encrypt both c_{L} and c'_{L} is non-observable, we assume that the adversary cannot decrypt either of the two ciphertexts himself. As such, due to the properties we require of the cipher, the adversary cannot distinguish between the two values. Note that the relation still holds if the two keys were non-equal, provided that they are both still unknown to the adversary.

3. $\mathbb{L} \vdash \mathrm{ctxt}(n, k_{\mathsf{L}}, c_{\mathsf{L}}) \not\approx_{\mathsf{L}} \mathrm{ctxt}(n', k'_{\mathsf{L}}, c_{\mathsf{L}})$, where $k_{\mathsf{L}} \neq k'_{\mathsf{L}}$

   In this example, the security level associated with both encryption keys means that the observer is permitted to obtain them. Consequently, we assume that the observer does indeed know both keys and is therefore able to correctly decrypt both ciphertexts. It then follows that, even though the plain texts are the same, the observer can distinguish between the two ciphertexts because a different key was required to correctly decrypt each one.

4. $\mathbb{L} \vdash \mathrm{ctxt}(n, k_{\mathsf{L}}, c_{\mathsf{L}}) \not\approx_{\mathsf{L}} \mathrm{ctxt}(n', k_{\mathsf{H}}, c_{\mathsf{L}})$, where $k_{\mathsf{L}} \neq k_{\mathsf{H}}$

   Here, the observer is only able to decrypt one of the two ciphertexts, and can thus deduce that the ciphertexts were created using different keys.

5. $\mathbb{L} \vdash \mathrm{ctxt}(n, k_{\mathsf{L}}, c_{\mathsf{H}}) \not\approx_{\mathsf{L}} \mathrm{ctxt}(n', k_{\mathsf{L}}, c'_{\mathsf{H}})$, where $c_{\mathsf{H}} \neq c'_{\mathsf{H}}$

   The two confidential constants in this example have been insecurely encrypted thus the observer can distinguish the two ciphertexts by first decrypting each one before comparing the resulting messages which have been leaked. This example is a further demonstration of the reason why the security levels associated with non-ciphertext values must be ignored: if they were not, then the two ciphertexts would incorrectly be deemed indistinguishable.

6. $\mathbb{L} \vdash \mathrm{ctxt}(n_1, k_{\mathsf{L}}, \mathrm{ctxt}(n_2, k_{\mathsf{H}}, c_{\mathsf{H}})) \approx_{\mathsf{L}} \mathrm{ctxt}(n_3, k_{\mathsf{L}}, \mathrm{ctxt}(n_4, k'_{\mathsf{H}}, c'_{\mathsf{H}}))$, where $c_{\mathsf{H}} \neq c'_{\mathsf{H}}$ and $k_{\mathsf{H}} \neq k'_{\mathsf{H}}$

The observer can correctly decrypt both ciphertexts using the same key, resulting in two further non-equal ciphertexts. However, because the observer is unable to distinguish between the two secondary ciphertexts, and both were obtained using the same decryption key, k_{L}, he has no way of distinguishing between the original ciphertexts.

Indistinguishability of Evaluation Domains

Definition 4.9 (Indistinguishability of Evaluation Environments)**.**

Two evaluation environments, Δ_1 *and* Δ_2*, are defined to be* indistinguishable *at the observation level l, written* $\Delta_1 \approx_l \Delta_2$*, when all of the following rules hold:*

- $\Delta_1 \equiv (\mathbb{L}, \mathbb{F}, \mathbb{N}_1, \phi_1)$
- $\Delta_2 \equiv (\mathbb{L}, \mathbb{F}, \mathbb{N}_2, \phi_2)$
- $\forall a : l' \in \mathbb{L} \;\; (l' \leq l) \rightarrow \mathbb{L} \vdash \phi_1(a) \approx_l \phi_2(a)$

Informally, this relation states that the observable components of the two evaluation environments are indistinguishable.

Definition 4.10 (Indistinguishability of Evaluation Contexts)**.**

Two evaluation contexts, Π_1 *and* Π_2*, are defined to be* indistinguishable *at the observation level l, written* $\Delta \vdash \Pi_1 \approx_l \Pi_2$*, when both of the following hold:*

- $\forall x \in \mathit{dom}(\Pi_1)\ ((x \mapsto v) \in \Pi_1 \wedge (lvl_\Delta(v) \leq l)) \rightarrow \exists v'.((x \mapsto v') \in \Pi_2 \wedge \Delta \vdash v \approx_l v')$
- $\forall x \in \mathit{dom}(\Pi_2)\ ((x \mapsto v') \in \Pi_2 \wedge (lvl_\Delta(v') \leq l)) \rightarrow \exists v.((x \mapsto v) \in \Pi_1 \wedge \Delta \vdash v \approx_l v')$

Informally, this relation states that the observable parts of the two evaluation contexts are indistinguishable. Consider the following examples:

1. $([(x \mapsto c_{\mathsf{L}})], \Delta) \not\approx_{\mathsf{L}} ([(x \mapsto c_{\mathsf{L}}), (y \mapsto c'_{\mathsf{L}})], \Delta)$

   The variable y, which maps to the observable value c'_{L}, has no corresponding variable in the other evaluation context.

2. $([(x \mapsto c_{\mathsf{L}})], \Delta) \approx_{\mathsf{L}} ([(x \mapsto c_{\mathsf{L}}), (y \mapsto c_{\mathsf{H}})], \Delta)$

   In this example, the variable y maps to the non-observable value c_{H}, therefore the two evaluation contexts have equal observable parts, and thus the two domains are considered indistinguishable.

3. $([(x \mapsto v_{\mathsf{L}}), (y \mapsto v'_{\mathsf{L}})], \Delta) \approx_{\mathsf{L}} ([(x \mapsto v''_{\mathsf{L}}), (y \mapsto v'''_{\mathsf{L}})], \Delta)$ iff $\mathbb{L} \vdash v_{\mathsf{L}} \approx_{\mathsf{L}} v''_{\mathsf{L}}$ and $\mathbb{L} \vdash v'_{\mathsf{L}} \approx_{\mathsf{L}} v'''_{\mathsf{L}}$

   The domain of variables which map to observable values is the same in both evaluation contexts, therefore the two evaluation domains are indistinguishable if and only if each pair of values are indistinguishable.

Now that we have formally defined what it means for values, evaluation contexts and evaluation states to be indistinguishable, we can now present the non-interference property that we will use:

Definition 4.11 (Non-Interferent Expression)**.**

An expression e is non-interferent *at the security level l with respect to two initial evaluation domains (Π_1, Δ_1) and (Π_2, Δ_2) when it follows from $\Delta_1 \approx_l \Delta_2$, $\Pi_1 \vdash \langle \Delta_1, e \rangle \Downarrow \langle \Delta'_1, v_1 \rangle$ and $\Pi_2 \vdash \langle \Delta_2, e \rangle \Downarrow \langle \Delta'_2, v_2 \rangle$ that $\mathbb{L}_{\Delta'_1} \vdash v_1 \approx_l v_2$ and $\Delta'_1 \approx_l \Delta'_2$.*

4.4 Type System

In this section, we present typing rules for the abstract system defined in §4.1, and then prove that they enforce the security properties from §4.3. We begin by defining the following types:

S	::=	$\langle \mathsf{E}, l \rangle$	*Security Type*
E	::=	$\mathsf{data} \mid \mathsf{bool} \mid l\ l\ \mathsf{key} \mid \mathsf{enc(S)} \mid \mathsf{S\ loc}$	*Data Types*
F	::=	$\{i : \mathsf{S}_i\ ^{i \in 1..n}\} \xrightarrow{l} \mathsf{S}$	*Function type*

The subtyping rules, which define the type hierarchy, are presented in Fig. 4.4. The security types are assigned to values as follows:

- $\langle \mathsf{data}, l \rangle$

  This type is assigned to any constant whose security level is no greater than l.

- $\langle \mathsf{bool}, l \rangle$

  This type is assigned to the Boolean values TRUE and FALSE, whose security

$$\frac{\mathsf{E} <: \mathsf{E}' \qquad l \leq l'}{\langle \mathsf{E}, l \rangle <: \langle \mathsf{E}', l' \rangle}\ \text{S-ST} \qquad \frac{\mathsf{S} <: \mathsf{S}'}{\mathsf{enc(S)} <: \mathsf{enc(S}')}\ \text{S-CTXT} \qquad \frac{}{\mathsf{T} <: \mathsf{T}}\ \text{S-REFL}$$

Figure 4.4: Subtyping rules in K

level is no greater than l. When applied to the result of an equality test, l will be no lower than the security level of each of the values being compared (see rule T-EQ in Fig. 4.5).

- $\langle l_1\ l_2\ \mathsf{key}, l_3 \rangle$
  This type is assigned to cryptographic keys, where the l_2 denotes the maximum security level of data which the key is permitted to encrypt, and l_1 specifies a lower bound on the security level of any sub-expression of an encrypted message (this corresponds to the minimum clearance level required to determine whether or not a decryption operation with that key has failed).

- $\langle \mathsf{enc(S)}, l \rangle$
  This type is assigned to a ciphertext containing an encrypted message of type S. Since messages can only be encrypted by a key whose security level is no less than that of the message, ciphertext may have any security level.

- $\langle \mathsf{S\ loc}, l \rangle$
  This type is associated with location identifiers for locations that may contain values of type S. We consider location identifiers to be public since knowledge of the identifier is not guaranteed to entail knowledge of the contents. As a result, l may be any security level.

  This type is similar to the τ **var** type for variables defined in [Volpano et al., 1996]. In Volpano, Smith and Irvine's type system, variables are read-only when given the type τ, and write-only when given the type τ **acc**. In K, variables are always read-only, and we can emulate read-only locations within an expression by assigning the value stored in that location to a variable, then referencing the variable instead (i.e., let $x = *a$ in e). Write-only variables are used by Volpano, Smith and Irvine to model function return values; in K, the return value of a function is the value to which the function body evaluates. Multiple return values can be achieved via passing in memory locations and assigning to them in the body of the function, as is done in the programming language C, for example.

For values, these assignments are defined formally by Lemma A.2 (Inversion of the Typing Relation for Values) and Lemma A.3 (Canonical Forms). The former maps from values to types, while the latter maps from types to values.

For any given security type, the security level denotes the confidentiality level of the associated expression. Lemma 4.19 shows that this confidentiality level is no lower

than the security level of the value that the expression evaluates to. Every expression has a *minimal type* which is dominated by all other types that may be associated with that term. This desirable property is not required to show any of the results in this thesis, so we do not give a proof.

To ensure that our type system only permits encryptions that produce ciphertext which may be declassified, it is necessary to place some restrictions upon the security levels in the type for cryptographic keys. Specifically, the security level associated with the key must be no lower than the security level associated with the messages it may encrypt. Additionally, since the remaining security level denotes a lower bound on the security levels contained within the message type, it must be no greater than the security level of the message. Any type which does not break these restrictions is termed a *valid type*, and the formal definition of this relation is as follows:

Definition 4.12 (Valid Security Types)**.**

A security type is defined to be valid *according to the following rules:*

- $\langle \mathsf{data}, l \rangle$ *is valid*
- $\langle \mathsf{bool}, l \rangle$ *is valid*
- $\langle \mathsf{S\ loc}, l \rangle$ *is valid iff* S *is valid*
- $\langle l_1\ l_2\ \mathsf{key}, l_3 \rangle$ *is valid iff* $l_1 \leq l_2 \leq l_3$
- $\langle \mathsf{enc(S)}, l \rangle$ *is valid iff* S *is valid*

The type for functions defines an upper bound on the security type of the input values, and a lower bound on the security type of the result value. The security level on the arrow denotes the minimum level to which information may flow as a result of that function being executed. For example, a function which takes high inputs, returns a high result, but modifies the contents of a memory location that can be read by anyone will have a type of the form $\{i : \langle \mathsf{E}_i, \mathsf{H} \rangle^{\ i \in 1..n}\} \xrightarrow{\mathsf{L}} \langle \mathsf{E}, \mathsf{H} \rangle$. This additional security level is required to track memory side-effects arising from calling the associated function.

Definition 4.13 (Minimum Level of a Security Type)**.**

The minimum level *of a security type* S*, denoted* $\lfloor \mathsf{S} \rfloor$*, is defined according to the following rules:*

- $\lfloor \langle \mathit{data}, l \rangle \rfloor = l$
- $\lfloor \langle \mathit{bool}, l \rangle \rfloor = l$
- $\lfloor \langle \mathsf{S}\ \mathit{loc}, l \rangle \rfloor = l \curlywedge \lfloor \mathsf{S} \rfloor$
- $\lfloor \langle l_1\ l_2\ \mathit{key}, l_3 \rangle \rfloor = l_1$
- $\lfloor \langle \mathit{enc}(\mathsf{S}), l \rangle \rfloor = l \curlywedge \lfloor \mathsf{S} \rfloor$

This relation returns the greatest lower bound of all security levels which are present in the type.

4.4.1 Typing Relation

Expressions are typed with respect to a *typing environment* Σ, *typing context* Γ and minimum write level l_w:

$$\Gamma \vdash_{l_w} e : \mathsf{S} \ \ \textit{w.r.t.}\ \ \Sigma \qquad\qquad \textit{Typing relation}$$

The typing context comprises a set of variable to type mappings, while the typing environment comprises types for constants, keys, locations and functions:

$$\begin{array}{ll} \Gamma\ ::=\ x : \mathsf{S}, \Gamma \mid \varepsilon & \textit{Typing context} \\ \Sigma\ ::=\ c : \mathsf{S}, \Sigma \mid k : \mathsf{S}, \Sigma \mid a : \mathsf{S}, \Sigma \mid f : \mathsf{F}, \Sigma \mid \varepsilon & \textit{Typing environment} \\ (\Gamma, \Sigma) & \textit{Typing domain} \end{array}$$

Definition 4.14 (Valid Typing Environment)**.**

A typing environment Σ *is defined to be* valid *when all of the security types present within it are valid.*

The minimum write is analogous to the program counter from Fenton's Data Mark Machine (as discussed in §2.2), and denotes the minimum confidentiality level of non-ciphertext values which the expression may evaluate to or write into memory locations.

Lemma A.5 shows that, for all non-ciphertext values, l_w is a lower-bound on the security level in the associated type. This highlights our intention that the only way for information to exist in a lower-security context is for it to be encrypted.

4.4.2 Domain Consistency

The security properties that we wish to prove are upheld by well-typed programs are defined independently of the type system. Consequently, to project the properties of a typed program onto the untyped evaluation, we must define a relationship between the typing and evaluation domains which formalises the notion that the typing domain is a valid representation of the evaluation domain:

Definition 4.15 (Domain Consistency)**.**

A typing domain (Γ, Σ) *and an evaluation domain* (Π, Δ) *are said to be* consistent*, written* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$*, precisely when the following conditions hold:*

- $c : \langle E, l \rangle \in \Sigma \quad \leftrightarrow \quad c : l \in \Delta$
- $k : \langle E, l \rangle \in \Sigma \quad \leftrightarrow \quad k : l \in \Delta$
- $a : \langle \langle E, l \rangle \, \mathsf{loc}, l' \rangle \in \Sigma \quad \leftrightarrow \quad a : l \in \Delta,\ (a \mapsto v) \in \Delta \ \textit{and} \ \vdash_{\perp} v : \langle E, l \rangle$
- $x : S \in \Gamma \quad \leftrightarrow \quad (x \mapsto v) \in \Pi \ \textit{and} \ \vdash_{\perp} v : S$
- $f : \{i : S_i \,^{i \in 1..n}\} \xrightarrow{l_w} S \in \Sigma \quad \leftrightarrow \quad f(x_i \,^{i \in 1..n}) = e \in \Delta,\ x_i \notin \mathsf{dom}(\Gamma) \ \textit{and}$ $(\Gamma, x_i : S_i) \vdash_{l_w} e : S$

For constants and keys, the security level in the corresponding type matches that given in the evaluation environment. For locations, the security level associated with the type of the content must match that given in the evaluation environment, and the value stored in that location must have the specified type. The type associated with a variable in the typing context must be valid for the value assigned to that variable in the evaluation environment. Finally, the type for each function in the typing domain must be valid with respect to the implementation given in the evaluation environment. Note that the requirement for each parameter variable to not be in the typing context can be satisfied easily via Convention 4.2 (Alpha Renaming).

Intuitively, this relation ensures that the security levels of the types in the typing domain are no lower than those in the evaluation domain. In other words, the typing domain cannot treat a value as being less secure than it actually is, but it may consider it to be more secure.

4.4.3 Typed API Definition

In Section 4.1, we presented the abstract syntax for API definitions. Recall that this definition maps constants, keys and locations to security levels, as well as defining the

functions which comprise the API. We now give a typed variant of this abstract syntax which maps constants, keys, locations and functions to security types:

$$d \;::=\; d\,;\, c : \langle \mathsf{data}, l\rangle \mid d\,;\, k : \langle l\; l\; \mathsf{key}, l\rangle \mid d\,;\, a : \mathsf{S} \mid d\,;\, l_w\; f(\overrightarrow{x : \mathsf{S}})\,\{e : \mathsf{S}\} \mid \varepsilon \qquad \textit{API definition}$$

Note that the type associated with a location defines the type of values which may be stored in that location, not the type of the location itself — in K, the type of the location will always be $\langle \mathsf{S}\,\mathsf{loc}, \bot\rangle$ since location identifiers are assumed to be public. The security level l_w associated with each API function defines the minimum write level when executing the function body, e. The security type S denotes the return type of the function. We obtain the untyped API definition from a given typed definition via the following erasure function, $[\![\cdot]\!]$:

$$\begin{aligned}
[\![d\,;\, c : \langle \mathsf{data}, l\rangle]\!] &= [\![d]\!]\,;\, c : l \\
[\![d\,;\, k : \langle l\; l\; \mathsf{key}, l\rangle]\!] &= [\![d]\!]\,;\, k : l \\
[\![d\,;\, a : \langle \mathsf{E}, l\rangle]\!] &= [\![d]\!]\,;\, a : l \\
[\![d\,;\, l_w\; f(\overrightarrow{x : \mathsf{S}})\,\{e : \mathsf{S}\}]\!] &= [\![d]\!]\,;\, f(\overrightarrow{x})\,\{e\} \\
[\![\varepsilon]\!] &= \varepsilon
\end{aligned}$$

4.4.4 Typing Rules

The typing and subtyping rules in K are shown in Figs. 4.5 and 4.4, respectively. We now discuss some of the more interesting type rules.

Primitive values are assigned a security type where the security level is no lower than the minimum write level, l_w. This reflects the fact that, when inside the branch of a conditional expression, the value carries with it information about the result of the associated guard expression.

In T-SDEC and T-IF, the minimum write level for the branch expressions are at least as great as the security level of the guard expression. An alternative approach in these two rules would have been to force the security level of the result to be at least as high as the level of the guard expression. However, this would prevent us from being able to return ciphertext with a lower security level in cases where that ciphertext will not reveal anything about the guard expression. Specifically, this alternative approach would not allow us to give the type $\langle \mathsf{enc}(\langle \mathsf{data}, \mathsf{H}\rangle), \mathsf{L}\rangle$ to the following expression:

$$\frac{c : \langle \mathsf{data}, l \rangle \in \Sigma}{\vdash_{l_w} c : \langle \mathsf{data}, l \curlyvee l_w \rangle}\ \textsc{T-Cnst} \qquad \frac{}{\vdash_{l} \textsc{true} : \langle \mathsf{bool}, l \rangle}\ \textsc{T-True}$$

$$\frac{k : \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle \in \Sigma \qquad l_1 \leq l_2 \leq l_3}{\vdash_{l_1} k : \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle}\ \textsc{T-Key} \qquad \frac{}{\vdash_{l} \textsc{false} : \langle \mathsf{bool}, l \rangle}\ \textsc{T-False}$$

$$\frac{a : \langle \mathsf{S\ loc}, l \rangle \in \Sigma \qquad l_w \leq \lfloor \mathsf{S} \rfloor}{\vdash_{l_w} a : \langle \mathsf{S\ loc}, l \curlyvee l_w \rangle}\ \textsc{T-Loc} \qquad \frac{l_w \leq \lfloor \langle \mathsf{E}, l \curlyvee l_w \rangle \rfloor}{x : \langle \mathsf{E}, l \rangle \vdash_{l_w} x : \langle \mathsf{E}, l \curlyvee l_w \rangle}\ \textsc{T-Var}$$

$$\frac{\Gamma \vdash_{l_w} e_1 : \mathsf{S}' \qquad x \notin \mathsf{dom}(\Gamma) \qquad (\Gamma, x : \mathsf{S}') \vdash_{l_w} e_2 : \mathsf{S}}{\Gamma \vdash_{l_w} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \mathsf{S}}\ \textsc{T-Let}$$

$$\frac{\vdash_{l_w} k : \langle l'\ l\ \mathsf{key}, \top \rangle \qquad \vdash_{l_w} v : \langle \mathsf{E}, l \rangle \qquad l' \leq \lfloor \langle \mathsf{E}, l \rangle \rfloor}{\vdash_{l_w} \mathsf{ctxt}(n, k, v) : \langle \mathsf{enc}(\langle \mathsf{E}, l \rangle), \bot \rangle}\ \textsc{T-Ctxt}$$

$$\frac{\Gamma \vdash_{l_w} e_k : \langle l'\ l\ \mathsf{key}, \top \rangle \qquad \Gamma \vdash_{l_w} e_m : \langle \mathsf{E}, l \rangle \qquad l' \leq \lfloor \langle \mathsf{E}, l \rangle \rfloor}{\Gamma \vdash_{l_w} \mathsf{senc}(e_k, e_m) : \langle \mathsf{enc}(\langle \mathsf{E}, l \rangle), \bot \rangle}\ \textsc{T-SEnc}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{l_w} e_k : \langle l'\ l_m\ \mathsf{key}, \top \rangle \qquad \Gamma \vdash_{l_w} e_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle \qquad l' \leq \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \\ x \notin \mathsf{dom}(\Sigma) \qquad (\Gamma, x : \langle \mathsf{E}_m, l_m \rangle) \vdash_{l' \curlyvee l_w} e_1 : \mathsf{S} \qquad \Gamma \vdash_{l' \curlyvee l_w} e_2 : \mathsf{S}\end{array}}{\Gamma \vdash_{l_w} \mathsf{try\ sdec}(e_k, e_c) = x\ \mathsf{in}\ e_1\ \mathsf{else}\ e_2 : \mathsf{S}}\ \textsc{T-SDec}$$

$$\frac{\Gamma \vdash_{l_w} e_1 : \langle \mathsf{E}, l \rangle \qquad \Gamma \vdash_{l_w} e_2 : \langle \mathsf{E}, l \rangle \qquad \mathsf{E} \not\equiv \mathsf{enc(S)}}{\Gamma \vdash_{l_w} e_1 == e_2 : \langle \mathsf{bool}, l \rangle}\ \textsc{T-Eq}$$

$$\frac{\Gamma \vdash_{l_w} e_1 : \langle \mathsf{bool}, l \rangle \qquad \Gamma \vdash_{l \curlyvee l_w} e_2 : \mathsf{S} \qquad \Gamma \vdash_{l \curlyvee l_w} e_3 : \mathsf{S}}{\Gamma \vdash_{l_w} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathsf{S}}\ \textsc{T-If}$$

$$\frac{\Gamma \vdash_{l_w} e_1 : \langle \mathsf{S\ loc}, \top \rangle \qquad \Gamma \vdash_{l_w} e_2 : \mathsf{S}}{\Gamma \vdash_{l_w} e_1 := e_2 : \mathsf{S}}\ \textsc{T-Asgn}$$

$$\frac{\Gamma \vdash_{l_w} e_1 : \langle \langle \mathsf{E}, l \rangle\ \mathsf{loc}, \top \rangle \qquad l_w \leq \lfloor \langle \mathsf{E}, l \curlyvee l_w \rangle \rfloor}{\Gamma \vdash_{l_w} *e : \langle \mathsf{E}, l \curlyvee l_w \rangle}\ \textsc{T-Dref}$$

$$\frac{f : \{ i : \mathsf{S}_i\ ^{i \in 1..n} \} \xrightarrow{l'_w} \langle \mathsf{E}, l \rangle \in \Sigma \qquad l_w \leq l'_w \qquad \forall i \in 1..n\ \ \Gamma \vdash_{l_w} e_i : \mathsf{S}_i}{\Gamma \vdash_{l_w} f(e_i\ ^{i \in 1..n}) : \langle \mathsf{E}, l \curlyvee l_w \rangle}\ \textsc{T-FnApp}$$

$$\frac{\Gamma' \vdash_{l'_w} e : \mathsf{S}' \qquad \Gamma' \subseteq \Gamma \qquad l_w \leq l'_w \qquad \mathsf{S}' <: \mathsf{S}}{\Gamma \vdash_{l_w} e : \mathsf{S}}\ \textsc{T-Sub}$$

Figure 4.5: Typing rules for expressions in K w.r.t. some fixed typing environment Σ

if (v_1) then	$v_1 : \langle \mathsf{bool}, \mathsf{H} \rangle$
$\quad \mathrm{senc}(k, v_2)$	$v_2 : \langle \mathsf{data}, \mathsf{H} \rangle$
else	$v_3 : \langle \mathsf{data}, \mathsf{H} \rangle$
$\quad \mathrm{senc}(k, v_3)$	$k : \langle \mathsf{H\,H\,key}, \mathsf{H} \rangle$

As can be seen in T-SENC (and T-CTXT), ciphertext is given the minimum security level, $\bot$, and T-IF does not increase this level based on l_w.

In T-SDEC, the inequality condition ensures that information about the success or otherwise of the decryption does not leak to a security level lower than l'. As shown by Lemma A.2 (Inversion of the Typing Relation for Values), we have $l_w \leq l'$ therefore the inequality also ensures that this information does not leak to a level lower than the minimum write level. This is a critical requirement for our proof of Theorem 4.21 (Non-Interference for Typed Expressions) to go through. Similar inequalities exist in other typing rules for this same reason.

The Bell-LaPadula $*$-security property ("no write down") is enforced in T-LOC by the inequality condition, while the rules do not directly enforce the simple security property ("no read-up"). The reason for the latter behaviour is that the adversary is only permitted to obtain the result of an evaluation and the contents of memory locations which may only contain values with an associated security level that is no greater than his own clearance level. In other words, for memory accesses, our model implicitly enforces no read up, while for return values, we are only interested in expressions which should return values that the adversary is permitted to read.

In T-EQ, the third condition is a consequence of Definition 4.8 (Indistinguishability of Values). This definition equates any two ciphertexts when the adversary is unable to decrypt either one. However, since two such ciphertexts may be non-equal, testing them for equality would subvert this weakening. An alternative approach would have been to modify E-EQ1 to return TRUE when the two values are indistinguishable, but that would detract from the intuitiveness of the evaluation semantics, so we decided against that option.

4.5 Security Proofs

In this section we prove that the evaluation of a well-typed client program which calls functions from a well-typed Security API is both confidentiality-preserving and non-interferent. The former property is implied by the latter, but we give a proof that is

independent of noninterference as it allows us to introduce many of the fundamental lemmas at an earlier stage.

For the sake of brevity and clarity, we omit many of the actual proofs from this chapter, providing only formal statements of the various supporting properties and lemmas, along with selected proof cases for the main theorems. The full proofs, for all properties, are given in Appendix A.

It should be noted that many proofs are only valid for *terminating* evaluations. This is due to the proof case for function application in Theorem 4.18 (Type Safety) requiring that all function definitions are sound with respect to that Theorem, and that is only true for terminating function calls. This restriction is reasonable as we do not consider side-channel attacks arising from non-termination in this thesis, since API functions are generally intended to always terminate, regardless of the supplied inputs.

To help simplify the function application sub-case in Theorem 4.18, we make the following assumption:

Assumption 4.16. *When the necessary preconditions exist, Theorem 4.18 (Type Safety) holds for all function body expressions defined in* Σ

This assumption does not restrict our result, as the termination requirement means that all function applications could be unrolled and replaced with the corresponding function body expression. Consequently, an equivalent program can be obtained that does not include any function calls, rendering this assumption redundant.

4.5.1 Confidentiality

Our first result — that well-typed expressions are confidentiality-preserving as defined by Definition 4.7 — is stated formally as follows:

Theorem 4.17 (Confidentiality Preservation)**.**

If $\Gamma \vdash_{l_w} e : \langle E, l \rangle$ *w.r.t.* Σ *then, for any valid evaluation domain* (Π, Δ) *such that* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$*, e is confidentiality-preserving with respect to* (Π, Δ) *and l.*

We prove this theorem via Type Safety — a standard theorem for type systems which states that the result of an evaluation will have the same type as the initial expression — and a lemma which provides an upper bound on the security level of a well-typed value:

Theorem 4.18 (Type Safety)**.**

For any valid typing domain (Γ, Σ) *and evaluation domain* (Π, Δ) *where* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$*, if* $\Gamma \vdash_{l_w} e : S$ *w.r.t.* Σ *then either* e *is a value, or there exists some final state* $\langle \Delta', v \rangle$ *such that* $\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta', v \rangle$*,* $(\Gamma, \Sigma) \simeq (\Pi, \Delta')$ *and* $\vdash_{l_w} v : S$ *w.r.t.* Σ*.*

Lemma 4.19 (Upper Bound on the Security Level of Well-Typed Values)**.**

For any valid typing domain (Γ, Σ) *and evaluation domain* (Π, Δ) *where* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$*, it follows from* $\vdash_{l_w} v : \langle E, l \rangle$ *w.r.t.* Σ *that* $lvl_\Delta(v) \leq l$*.*

The proofs for Theorem 4.18 and Lemma 4.19 are both relatively straightforward, although they are dependent upon Definition 4.15 (Domain Consistency) — this is why the Type Safety Theorem also requires that this relationship is maintained.

Given these two properties, the proof of Theorem 4.17 (Confidentiality Preservation) is reasonably straightforward:

Proof.

By Theorem 4.18 (Type Safety), either e is a value or we have $(\Gamma, \Sigma) \simeq (\Pi, \Delta')$ and $\vdash_{l_w} v : \langle \mathsf{E}, l \rangle$. In the former case, we have $v = e$ and $\Delta' = \Delta$. We already have $\Gamma \vdash_{l_w} v : \langle \mathsf{E}, l \rangle$ and since values do not contain variables, we get $\vdash_{l_w} v : \langle \mathsf{E}, l \rangle$.

Consequently, in both cases, we get $lvl_\Delta(v) \leq l$ via Lemma 4.19 (Upper Bound on the Security Level of Well-Typed Values).

We also have $(\Gamma, \Sigma) \simeq (\Pi, \Delta')$ in both cases, therefore it follows from Definition 4.15 (Domain Consistency) that, for all $a : l' \in \Delta'$, we have $\vdash_\perp v' : \langle \mathsf{E}', l' \rangle$ where $a \mapsto v' \in \Delta'$ and $a : \langle \langle \mathsf{E}', l' \rangle \, \mathsf{loc}, l'' \rangle \in \Sigma$. It therefore follows from Lemma 4.19 (Upper Bound on the Security Level of Well-Typed Values) that for each v' we have $lvl_{\Delta'}(v') \leq l'$. □

4.5.2 Non-Interference

Our second result — that well-typed expressions are non-interferent — is stated formally as follows:

Theorem 4.20 (Non-Interferent Expression)**.**

If $\Gamma \vdash_{l_w} e : \langle E, l \rangle$ *w.r.t.* Σ *then, for any two evaluation domains* (Π_1, Δ_1) *and* (Π_2, Δ_2) *such that* $(\Gamma, \Sigma) \simeq (\Pi_1, \Delta_1)$*,* $(\Gamma, \Sigma) \simeq (\Pi_2, \Delta_2)$*,* $\Gamma \vdash \Pi_1 \approx_{l_o} \Pi_2$ *and* $\Sigma \vdash \Delta_1 \approx_{l_o} \Delta_2$ *where* $l \leq l_o \leq l_w$*, e is non-interferent at security level* l_o *with respect to* (Π_1, Δ_1) *and* (Π_2, Δ_2)*.*

From a security perspective, we are interested in cases where l_o is the observation level of the adversary, and the adversary is permitted to obtain the result of the evaluation,

thus we will have $l \leq l_o$. Since l_w denotes the maximum level to which no downward information flows are permitted by the type system, it follows that the expression can only be non-interferent when $l_w \geq l_o$. Typically, l and l_o will be $\bot$, in which case l_w can be any level.

We prove this theorem by first showing that well-typed expressions uphold a typed version of indistinguishability (see Definition A.6), then subsequently show that, given the preconditions of our second result, this typed version of indistinguishability implies the untyped version which we require.

Theorem 4.21 (Non-Interference for Typed Expressions)**.**

If $\Gamma \vdash_{l_w} e : S$ *w.r.t.* Σ, $\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta_r, v \rangle$ *and* $\Pi' \vdash \langle \Delta', e \rangle \Downarrow \langle \Delta'_r, v' \rangle$, *where* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$, $(\Gamma, \Sigma) \simeq (\Pi', \Delta')$, Σ *is valid,* $\Gamma \vdash \Pi \approx_{l_o} \Pi'$, $\Sigma \vdash \Delta \approx_{l_o} \Delta'$ *and* $l_o \leq l_w$, *then* $v \approx_{l_o} v' : S$ *and* $\Sigma \vdash \Delta_r \approx_{l_o} \Delta'_r$.

Lemma 4.22 (Indistinguishability)**.**

If $\vdash_{l_w} v_1 : \langle E, l \rangle$ *w.r.t.* Σ, $\vdash_{l_w} v_2 : \langle E, l \rangle$ *w.r.t.* Σ *and* $v_1 \approx_{l_o} v_2 : \langle E, l \rangle$ *then, for any typing context* Γ *and evaluation domain* (Π, Δ) *where* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$, *it follows from* $l_o \geq l$ *that* $\Delta \vdash v_1 \approx_{l_o} v_2$.

Lemma 4.23 (Indistinguishability of Evaluation Environments)**.**

If $\Sigma \vdash \Delta_1 \approx_l \Delta_2$, $(\Gamma, \Sigma) \simeq (\Pi_1, \Delta_1)$ *and* $(\Gamma, \Sigma) \simeq (\Pi_2, \Delta_2)$ *then* $\Delta_1 \approx_l \Delta_2$.

Theorem 4.21 is the critical result which proves that our type system enforces non-interference. The proof itself relies upon a key Lemma, and this is discussed further in the next section, where we present the case for the decryption rule (the proof itself proceeds via structural induction on the form of the typing derivation).

Given the three properties from above, the proof of Theorem 4.20 (Non-Interferent Expression) proceeds as follows:

Proof.

For any two evaluations of the form $\Pi_1 \vdash \langle \Delta_1, e \rangle \Downarrow \langle \Delta'_1, v_1 \rangle$ and $\Pi_2 \vdash \langle \Delta_2, e \rangle \Downarrow \langle \Delta'_2, v_2 \rangle$, it follows from Theorem 4.21 (Non-Interference for Typed Expressions) that $v_1 \approx_l v_2 : \langle \mathsf{E}, l \rangle$ and $\Sigma \vdash \Delta'_1 \approx_l \Delta'_2$. From Theorem 4.18 (Type Safety), we get $\vdash_{l_w} v_1 : \langle \mathsf{E}, l \rangle$, $\vdash_{l_w} v_2 : \langle \mathsf{E}, l \rangle$, $(\Gamma, \Sigma) \simeq (\Pi_1, \Delta'_1)$ and $(\Gamma, \Sigma) \simeq (\Pi_2, \Delta'_2)$. The first part of our result, $\mathbb{L}_{\Delta'_1} \vdash v_1 \approx_{l_o} v_2$, follows via Lemma 4.22 (Indistinguishability), while the second part, $\Delta'_1 \approx_{l_o} \Delta'_2$ follows via Lemma 4.23 (Indistinguishability of Evaluation Environments). □

4.5.3 Proving Non-Interference for Typed Expressions

The full proof of Theorem 4.21 (Non-Interference for Typed Expressions) is given in §A.2.3, and proceeds via structural induction on the form of the typing derivation. For those type rules which do not correspond to expressions that can affect program control-flow, the associated proof cases generally follow from the relevant definitions and Theorem 4.18 (Type Safety). The remaining two cases, for decryption (T-SDEC) and conditional branching (T-IF), both rely on the following Lemma:

Lemma 4.24 (Typed Indistinguishability of Arbitrary Values)**.**

If $\vdash_{l_w} v : S$ *w.r.t.* Σ, $\vdash_{l_w} v' : S$ *w.r.t.* Σ, Σ *is valid and* $l < l_w$ *then* $v \approx_l v' : S$.

This Lemma states that, for any two values which can both be given the same type at the same minimum write level, those values will be indistinguishable at that type for all observation levels less than the minimum write level.

Corollary 4.25. *If* $\vdash_{l_w} v : S$ *w.r.t.* Σ, $\vdash_{l_w} v' : S$ *w.r.t.* Σ, Σ *is valid and* $v \not\approx_l v' : S$ *then* $l \not< l_w$.

In other words, the minimum write level denotes a lower-bound on the clearance level required to obtain information about any two values with the same type via well-typed expressions. In conjunction with the restrictions that are placed upon the minimum write level of branch expressions (via T-SDEC and T-IF), Lemma 4.24 shows that the values returned by each branch are indistinguishable when the observer is unable to see the outcome of the guard expression.

We will now show how this result is used in the proof case for decryption, which is the most complex case in the entire proof of Theorem 4.21. First, we identify what must hold when each evaluation has to choose which branch to take, then analyse each of the two sub-cases corresponding to the same or different branch being chosen.

By inspection of rule T-SDEC, we have:

$$e \;=\; \text{try } \mathrm{sdec}(e_k, e_c) = x \text{ in } e_1 \text{ else } e_2$$

$$\Gamma \vdash_{l_w} e_k : \langle l'\ l_m\ \mathsf{key}, \top \rangle$$

$$\Gamma \vdash_{l_w} e_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle$$

$$l' \leq \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor$$

$$x \notin \mathsf{dom}(\Sigma)$$

$$(\Gamma, x : \langle \mathsf{E}_m, l_m \rangle) \vdash_{l' \curlyvee l_w} e_1 : \mathsf{S}$$

$$\Gamma \vdash_{l' \curlyvee l_w} e_2 : \mathsf{S}$$

By inspection of the evaluation rules, either E-SDEC1 or E-SDEC2 could have been used at the root of each evaluation. In both cases, the key and ciphertext expressions are evaluated in that order, therefore we must have:

$$\Pi \vdash \langle \Delta, e_k \rangle \Downarrow \langle \Delta_1, k \rangle \qquad \Pi' \vdash \langle \Delta', e_k \rangle \Downarrow \langle \Delta'_1, k' \rangle$$

$$\Pi \vdash \langle \Delta_1, e_c \rangle \Downarrow \langle \Delta_2, v_c \rangle \qquad \Pi' \vdash \langle \Delta'_1, e_c \rangle \Downarrow \langle \Delta'_2, v'_c \rangle$$

By multiple applications of Theorem 4.18 (Type Safety):

$$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1) \qquad (\Gamma, \Sigma) \simeq (\Pi', \Delta'_1)$$

$$(\Gamma, \Sigma) \simeq (\Pi, \Delta_2) \qquad (\Gamma, \Sigma) \simeq (\Pi', \Delta'_2)$$

$$\vdash_{l_w} v_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle \qquad \vdash_{l_w} v'_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle$$

By two applications of the induction hypothesis, we get:

$$k \approx_{l_o} k' : \langle l'\ l_m\ \mathsf{key}, l \rangle \qquad v_c \approx_{l_o} v'_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), l \rangle$$

$$\Sigma \vdash \Delta_1 \approx_{l_o} \Delta'_1 \qquad \Sigma \vdash \Delta_2 \approx_{l_o} \Delta'_2$$

By Lemma A.3 (Canonical Forms):

$$v_c = \mathrm{ctxt}(n, k_1, v_m)$$

$$v'_c = \mathrm{ctxt}(n', k_2, v'_m)$$

By Definition A.6 (Typed Indistinguishability of Values):

$$k = k' \quad or \quad l_o < l'$$

$$k_1 = k_2 \quad or \quad l_o < l'$$

We now split on the relative ordering of l_o and l':

- Subcase $l_o < l'$:

  By the properties of $\curlyvee$:

  $$l_o < (l' \curlyvee l_w)$$

  By Theorem 4.18 (Type Safety):

  $$\vdash_{l_w} k : \langle l'\ l_m\ \mathsf{key}, \top \rangle$$

  In this subcase, each evaluation may have E-SDEC1 or E-SDEC2 at its root, giving us:

  $$\begin{array}{lll} v & \in & \{v_1, v_2\} \\ v' & \in & \{v'_1, v'_2\} \end{array} \quad \text{where} \quad \begin{array}{l} \Pi \cup (x \mapsto v_m) \vdash \langle \Delta_2, e_1 \rangle \Downarrow \langle \Delta_r, v_1 \rangle \\ \Pi \vdash \langle \Delta_2, e_2 \rangle \Downarrow \langle \Delta_r, v_2 \rangle \\ \Pi' \cup (x \mapsto v'_m) \vdash \langle \Delta'_2, e_1 \rangle \Downarrow \langle \Delta'_r, v_2 \rangle \\ \Pi' \vdash \langle \Delta'_2, e_2 \rangle \Downarrow \langle \Delta'_r, v'_2 \rangle \end{array}$$

  By Definition 4.15 (Domain Consistency):

  $$((\Gamma, x : \langle \mathsf{E}_m, l_m \rangle), \Sigma) \simeq (\Pi \cup (x \mapsto v_m), \Delta_2)$$
  $$((\Gamma, x : \langle \mathsf{E}_m, l_m \rangle), \Sigma) \simeq (\Pi' \cup (x \mapsto v'_m), \Delta'_2)$$

  By two applications of Theorem 4.18 (Type Safety):

  $$\vdash_{l' \curlyvee l_w} v : \mathsf{S}$$
  $$\vdash_{l' \curlyvee l_w} v' : \mathsf{S}$$

  By Lemma 4.24 (Typed Indistinguishability of Arbitrary Values):

  $$\underline{v \approx_{l_o} v' : \mathsf{S}}$$

  By two separate applications of Lemma A.12 (Preservation Under Evaluation of Typed Indistinguishability for Evaluation Environments):

  $$\Sigma \vdash \Delta_2 \approx_{l_o} \Delta_r$$
  $$\Sigma \vdash \Delta'_2 \approx_{l_o} \Delta'_r$$

  By two applications of Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

  $$\underline{\Sigma \vdash \Delta_r \approx_{l_o} \Delta'_r}$$

- Subcase $l_o \not\preceq l'$:

  In this subcase, we must have:

  $$\left.\begin{array}{l} k = k' \\ k_1 = k_2 \end{array}\right\} \Rightarrow \quad k = k_1 \text{ iff } k' = k_2$$

  As a result, both evaluations must have the same rule at their root, giving us:

  $$\begin{array}{ll} \Pi \cup (x \mapsto v_m) \vdash \langle \Delta_2, e_1 \rangle \Downarrow \langle \Delta_r, v \rangle & \Pi \vdash \langle \Delta_2, e_2 \rangle \Downarrow \langle \Delta_r, v \rangle \\ \Pi' \cup (x \mapsto v'_m) \vdash \langle \Delta'_2, e_1 \rangle \Downarrow \langle \Delta'_r, v' \rangle & \Pi' \vdash \langle \Delta'_2, e_2 \rangle \Downarrow \langle \Delta'_r, v' \rangle \end{array} \quad \textit{or}$$

  By Definition A.8 (Typed Indistinguishability of Evaluation Contexts):

  $$(\Gamma, x : \langle \mathsf{E}_m, l_m \rangle) \vdash \Pi \cup (x \mapsto v_m) \approx_{l_o} \Pi' \cup (x \mapsto v'_m)$$

  Consequently, for each of the two possible outcomes, the result follows from the induction hypothesis. □

4.6 Ill-Typed Expressions

As noted at the beginning of Chapter 3, our type system may not prevent attacks that arise from the adversary program being ill-typed. Examples of such attacks are given in Fig. 4.6.

In the first attack, the adversary has access to the secret value, while in the second, he reads the memory location where it is stored. In the third attack, the API function encrypt_secret_1 expects to be given a secret key, but the adversary passes in a known key, while in the fourth attack, the API function encrypt_secret_2 expects to be given a location that stores a secret key, but the adversary passes in a location that stores a known key. In the final two cases, the adversary is able to decrypt the returned ciphertext and obtain the secret value. However, these types of attack should never arise in practice, for the following reasons:

- We assume that the adversary does not begin with knowledge of any secret terms, and therefore cannot directly use any of them in the programs he may run. This rules out the first attack.

- We assume that the adversary cannot directly obtain the contents of memory locations which may store secret values. For security APIs provided by Hardware Security Modules, such locations exist within the tamper-proof enclosure,

secret : ⟨data, ⊤⟩ *// secret value*

// Encrypts the secret value with the given (hopefully secret!) key
⟨enc(⟨data, ⊤⟩), ⊥⟩
encrypt_secret_1(*key* : ⟨⊥ ⊤ key, ⊤⟩) {
  senc(*key*, secret)
}

// Encrypts the secret value with the (hopefully secret!) key stored in the given
// location
⟨enc(⟨data, ⊤⟩), ⊥⟩
encrypt_secret_2(*hKey* : ⟨⟨⊥ ⊤ key, ⊤⟩ loc, ⊥⟩) {
  senc(**hKey*, secret)
}

// Adversary programs, which evaluate to secret
$\mathbb{L} = \{\text{key} : \bot, a_1 : \bot, a_2 : \top\}$
$\phi = \{a_1 \mapsto \text{key}, a_2 \mapsto \text{secret}\}$

e_1 = secret
e_2 = $*a_2$
e_3 = try sdec(key, encrypt_secret(key)) = x in x else FALSE
e_4 = try sdec(key, encrypt_secret(a_1)) = x in x else FALSE

Figure 4.6: Type-confusion attacks on a well-typed API.

while for software-based security APIs, access to such locations are restricted at compile-time or by the operating system. This rules out the second attack.

- API functions should accept as inputs only those types for values which the intended callers are expected to know. For example, API functions which any user may call should accept only public inputs, while API calls that security officers may call could accept higher security inputs. This rules out the third attack as encrypt_secret_1 expects a secret key.

- API functions which accept locations that are expected to contain secret values should verify that the provided location does indeed refer to such a location. In practice, a security API will know which memory locations may be accessed by

the caller, and which may not (e.g., based on memory address ranges). This rules out the fourth attack as encrypt_secret_2 would reject the supplied location.

We could have enforced these restrictions in our evaluation rules, but it would have complicated the proofs and would not have prevented the adversary from providing ill-typed inputs to API calls (e.g., using arbitrary data where ciphertext or a key is expected). Providing security guarantees in the presence of ill-typed adversary code is left as future work.

For now, our type system can be used to ensure that an API is secure when used as intended, or when client code can be type-checked in advance (e.g., software-based security APIs). For example, PKCS #11 has been shown to leak sensitive data when certain sequences of well-typed API calls are made [Clulow, 2003b], [Tsalapati, 2007], [Delaune et al., 2008], [Centenaro et al., 2012].

In the next chapter, we show how our type system can be used to provide security guarantees for a restricted implementation of PKCS #11 that disallows these insecure sequences of API calls.

Chapter 5

Example API Model

In this chapter, we demonstrate how our system can be used to model the cryptographic API described in [Cachin and Chandran, 2009]. We have chosen this particular API because it covers core functionality from real-world security APIs, and the authors prove that it enforces specific security policies.

In presenting this model, we show that the enforcement of these security policies follow automatically as a consequence of the model being well-typed in our system. This serves to highlight the utility of our type system as a tool for enforcing practical security policies in real-world APIs without the need for custom proofs.

5.1 API Overview

In [Cachin and Chandran, 2009], the authors present a formal model of a multi-user security API with an explicitly defined security policy, comprising common functions such as encryption and decryption, key wrapping and unwrapping, and cryptographic signing and verification of data. In this API, each key has an Access Control List (ACL) defining which operations each user may use that key for, along with additional usage restrictions that are required in order for the security properties to be upheld. These ACLs, along with a record containing information about what operations each key has previously been used for, provide the basis for various checks that each API function must carry out in order for that function to execute successfully.

The API is shown to be capable of representing a subset of PKCS #11 v2.20 [PKCS #11] which covers encryption and decryption, signing and verification, and various functions related to key management. Moreover, the authors prove that the standard PKCS #11 security policies are enforced by this subset.

5.1.1 Restrictions

The system presented in this thesis is not capable of modelling the entirety of Cachin and Chandran's API as it does not cater for public/private key pairs nor cryptographic signing and verification. Additionally, as our system enforces security through static typing, we cannot model the creation and/or deletion of users or keys nor the explicit modification of ACLs. We therefore must make the following restrictions:

1. All keys are for use with symmetric encryption algorithms
2. The set of keys is fixed
3. The set of users is fixed
4. The ACL associated with each key is fixed
5. The ACL associated with each key only includes the *read* privilege

These restrictions do not preclude our ability to enforce the same security properties as Cachin and Chandran, and all but the latter are enforced by many APIs for regular (non-administrator) users. We only consider the *read* privilege explicitly, as the other privileges are either modelled via types (see §5.2.1), or apply to key usages which we cannot model (e.g., key derivation). Our model will comprise the following functions: read, encrypt, decrypt, wrap and unwrap.

A final difference between our model and that of Cachin and Chandran is that they lazily enforce usage restrictions on a key. That is, a key may be used the first time for any cryptographic operation permitted by its ACL, but subsequent usages depend upon what it was previously used for. For example, a key used to wrap another key cannot subsequently be used to decrypt public data. In our system however, as a consequence of using static typing, the permitted usages of a key must be specified in advance (via the type assigned to it).

One outcome of the model we present in this chapter is that the type of a key in our system is sufficient to satisfy the rules specified by Cachin and Chandran, so we remove the need to carry out runtime checking of a key's previous uses.

5.1.2 Key Attributes

In Cachin and Chandran's API, each cryptographic key has an associated collection of attributes and an ACL which defines what operations each user may use that key

with. For each key, the API also keeps track of its *dependents* and *readers*. A key k_1 is dependent upon another key k_2 if $k_1 = k_2$, or k_2 has been used to wrap k_1, or there exists a third key k_3 such that k_1 is dependent upon k_3 and k_3 is dependent upon k_2. The readers of a key k_1 are those users who have used $\mathsf{read}(k)$ to obtain the cryptographic value of any key upon which k_1 is dependent. Due to the restrictions outlined above, our model only has to consider the *unextractable* attribute along with the readers of each key.

5.1.3 Security Policy

Cachin and Chandran specify the following security properties that their cryptographic API upholds:

1. A user may only obtain the cryptographic value of a key whose ACL permits that user to read the key.

   "[T]he adversary may never add the read privilege to the ACL of any key ... without this privilege, in particular, the key cannot become known to the adversary" (p. 16, ¶ 1)

2. A user must not be able to distinguish between two pieces of ciphertext (either directly or via API calls) when that user is not permitted to obtain the plaintexts within.

   "The adversary may try to break the security of the token ... by distinguishing legitimate encryptions from encryptions of a dummy message" (p. 16, ¶ 2)

3. An *unextractable* key must remain inside the cryptographic device and therefore cannot be read or wrapped.

   "Keys with the unextractable attribute must remain on the token forever, and cannot be read or wrapped" (p. 13, ¶ 6)

4. An *unextractable* key cannot have its ACL modified.

   "Keys with the unextractable attribute ... its ACL can no longer be modified" (p. 13, ¶ 6)

5.2 API Model

We now describe our model of Cachin and Chandran's cryptographic API along with the various design decisions that were made.

5.2.1 Security Types

Cachin and Chandran enforce their security policy by way of key attributes, ACLs and a key usage record, and it is our aim to show that the same security policy can be enforced through our type system alone. To this end, we construct a security level lattice that represents key attributes and ACLs, and show that our type system makes redundant the key usage record when used with this lattice. We begin by observing some properties that the types of keys in our model must possess:

1. Our type system requires that the security level of a key is at least as great as the security level of any value it encrypts. As a result, *the security level associated with any key-wrapping key must be greater than the security level associated with any data-encrypting key*.

2. Cachin and Chandran's API allows a key to be used to wrap another key or to encrypt messages, but not both. It is therefore necessary that *the type of any key-encrypting key and the type of any data-encrypting key are incomparable*. However, our type system requires that the security level of a key-encrypting key must dominate the security level of the key it encrypts, therefore this property cannot be enforced in the types if we want to allow for the wrapping of data-encryption keys. Consequently, this property will have to be enforced via the API function signatures.

3. An *unextractable* key cannot be wrapped by a key-encrypting key. Consequently, *the security level associated with an unextractable key must be greater than the security level associated with any key-encrypting key*.

4. An *unextractable* key cannot be read by any user. Consequently, *the security level associated with an unextractable key must be greater than the security level associated with any key that has the 'read' privilege for some user*.

5. Our type system allows for the arbitrary increase of the security level associated with a value. With regards to the ACL of a key, this means that *a higher security level must represent a more restrictive ACL.*

We can obtain these properties by using the following security level lattice:

$$
\begin{array}{rcll}
l & ::= & (\mathcal{X}, \mathcal{T}, \mathcal{R}) & \textit{security level} \\
\mathcal{X} & ::= & \mathsf{x} \mid \overline{\mathsf{x}} & \textit{extractable/unextractable} \\
\mathcal{T} & ::= & \mathsf{d} \mid \mathsf{k} & \textit{data/key} \\
\mathcal{R} & \subseteq & \mathcal{U} & \textit{readers}
\end{array}
$$

$$
\frac{\mathcal{X} \leq \mathcal{X}' \qquad \mathcal{R}' \subseteq \mathcal{R}}{(\mathcal{X}, \mathcal{T}, \mathcal{R}) \leq (\mathcal{X}', \mathcal{T}, \mathcal{R}')}\ \text{L-SecLev} \qquad \frac{}{\mathsf{d} \leq \mathsf{k}}\ \text{L-Type} \qquad \frac{}{\mathsf{x} \leq \overline{\mathsf{x}}}\ \text{L-Ext}
$$

It therefore follows that:

$$
\begin{array}{rcll}
\top & = & (\overline{\mathsf{x}}, \mathsf{k}, \emptyset) & \\
\bot & = & (\mathsf{x}, \mathsf{d}, \mathcal{U}) & \\
l_u & = & (\mathsf{x}, \mathsf{k}, \{u\}) & \textit{Clearance level of logged-in user } u
\end{array}
$$

Given this security level lattice, we assign types to keys and arbitrary data as follows,:

$$
\left.
\begin{array}{rcl}
\text{Arbitrary Data} & : & \langle \mathsf{data}, (\mathsf{x}, \mathsf{d}, \mathcal{R}) \rangle \\
\text{Data-Encrypting Keys} & : & \langle l\ (\mathsf{x}, \mathsf{d}, \mathcal{R}')\ \mathsf{key}, (\mathcal{X}, \mathsf{k}, \mathcal{R}) \rangle \\
\text{Key-Encrypting Keys} & : & \langle l\ (\mathsf{x}, \mathsf{k}, \mathcal{R}')\ \mathsf{key}, (\mathcal{X}, \mathsf{k}, \mathcal{R}) \rangle \\
\text{Key Handles} & : & \langle \langle l\ l'\ \mathsf{key}, l'' \rangle\ \mathsf{loc}, (\mathsf{x}, \mathsf{d}, \mathcal{R}) \rangle
\end{array}
\right\} \textit{where } (\mathcal{X} = \overline{x}) \rightarrow (\mathcal{R} = \emptyset)
$$

In all cases, $\mathcal{R}$ denotes those users who are permitted to know the corresponding value, and $\mathcal{R}' \subseteq \mathcal{R}$. The key types encode combinations of attributes that are permitted by Cachin and Chandran's setattr function.

5.2.2 Function Definitions

We now present implementations of the five API functions that our model comprises:[1]

read(k)

l
$\langle l\ l'\ \mathsf{key}, (\mathsf{x}, \mathsf{k}, \{u\}) \rangle$
read($hKey : \langle \langle l\ l'\ \mathsf{key}, (\mathsf{x}, \mathsf{k}, \mathcal{R}) \rangle\ \mathsf{loc}, (\mathsf{x}, \mathsf{d}, \{u\}) \rangle$) {
  $*hKey$
}

[1] In each implementation, the first line defines the minimum write-level and the second line defines the return type. Where present, instances of u correspond to the user that is calling the function.

In order for the read operation to execute successfully when the specified key handle is associated with a symmetric key, Cachin and Chandran require that the user which calls this function must be permitted to obtain the cryptographic value of any and all dependents of that key. For this function to be well-typed, we must have $u \in \mathcal{R}$. Since our type system forces l' to be dominated by $(\mathsf{x}, \mathsf{k}, \mathcal{R})$, it follows that l' is of the form $(\mathsf{x}, \mathcal{T}, \mathcal{R}')$ where $\mathcal{R} \subseteq \mathcal{R}'$. Consequently, when the specified key is a key-encrypting key (i.e., $\mathcal{T} = \mathsf{k}$), all dependents must have a security level of the form $(\mathsf{x}, \mathcal{T}, \mathcal{R}'')$ where $u \in \mathcal{R}''$, thus the check will pass.

Cachin and Chandran do not explicitly state that the key whose value is to be read must be extractable, but their setattr function requires that unextractable keys have an ACL where no users are permitted to read the cryptographic value. As expected, it therefore follows that any key whose ACL permits the cryptographic value to be read must be extractable.

encrypt(k,m)

l
$\langle \mathsf{enc}(\langle \mathsf{E}, l' \rangle), \bot \rangle$
encrypt($hKey : \langle \langle l\ (\mathsf{x}, \mathsf{d}, \mathcal{R})\ \mathsf{key}, (\mathcal{X}, \mathsf{k}, \mathcal{R}') \rangle\ \mathsf{loc}, (\mathsf{x}, \mathsf{d}, \{u\}) \rangle, msg : \langle \mathsf{E}, l' \rangle$) {
  senc(*$hKey, msg$)
}

Cachin and Chandran specify three checks that must pass for the encrypt operation to execute successfully:

1. The type of the encryption key is *symmetric* or *public*. Since our model only considers symmetric keys, this check will pass.

2. The encryption key has an ACL which permits the calling user to use that key with the encrypt operation. In our model, this restriction is enforced by the second parameter in the key type tuple.

3. The encryption key has only ever been used in the encrypt or decrypt operations. The security level of the operand for the unwrapping key is specified as $(\mathsf{x}, \mathsf{d}, \mathcal{R})$ therefore it follows from the forms of types for keys in this model that only data-encrypting keys can be given this type. As a result, this check will pass.

decrypt(k,c,r)

$$
\begin{array}{l}
l \\
\langle \mathsf{bool}, l \rangle \\
\mathrm{decrypt}(hKey : \langle\langle l\ (\mathsf{x}, \mathsf{d}, \mathcal{R})\ \mathsf{key}, (\mathcal{X}, \mathsf{k}, \mathcal{R}')\rangle\ \mathsf{loc}, (\mathsf{x}, \mathsf{d}, \{u\})\rangle, \\
\qquad ctxt : \langle \mathsf{enc}(\langle \mathsf{E}, l' \rangle), (\mathsf{x}, \mathsf{d}, \{u\})\rangle, \\
\qquad pRes : \langle\langle \mathsf{E}, l'' \rangle\ \mathsf{loc}, (\mathsf{x}, \mathsf{d}, \{u\})\rangle)\ \{ \\
\quad \mathrm{try}\ msg = \mathrm{sdec}(*hKey, ctxt)\ \mathrm{in} \\
\qquad \mathrm{let}\ x = (pRes := msg)\ \mathrm{in}\ \mathrm{TRUE} \\
\quad \mathrm{else} \\
\qquad \mathrm{FALSE} \\
\}
\end{array}
$$

Cachin and Chandran specify three checks that must pass for the decrypt operation to execute successfully:

1. The type of the decryption key is *symmetric* or *private*. Since our model only considers symmetric keys, this check will pass.

2. The decryption key has an ACL which permits the calling user to use that key with the decrypt operation. In our model, this restriction is enforced by the second parameter in the key type tuple.

3. The decryption key has only ever been used in the encrypt or decrypt operations. The security level of the operand for the unwrapping key is specified as $(\mathsf{x}, \mathsf{d}, \mathcal{R})$ therefore it follows from the forms of types for keys in this model that only data-encrypting keys can be given this type. As a result, this check will pass.

wrap(k1,k2)

$$
\begin{array}{l}
l \\
\langle \mathsf{enc}(\langle l'_1\ l'_2\ \mathsf{key}, (\mathsf{x}, \mathsf{k}, \mathcal{R}'')\rangle), \bot\rangle \\
\mathrm{wrap}(hKey1 : \langle\langle l_1\ (\mathsf{x}, \mathsf{k}, \mathcal{R}')\ \mathsf{key}, (\mathsf{x}, \mathsf{k}, \mathcal{R})\rangle\ \mathsf{loc}, (\mathsf{x}, \mathsf{d}, \{u\})\rangle, \\
\qquad hKey2 : \langle\langle l'_1\ l'_2\ \mathsf{key}, (\mathsf{x}, \mathsf{k}, \mathcal{R}'')\rangle\ \mathsf{loc}, (\mathsf{x}, \mathsf{d}, \{u\})\rangle)\ \{ \\
\quad \mathrm{senc}(*hKey1, *hKey2) \\
\}
\end{array}
$$

Cachin and Chandran specify seven checks that must pass for the wrap operation to execute successfully:

1. The type of the wrapping key is *symmetric*. As noted earlier, our model only considers symmetric keys, therefore this check will pass.

2. The wrapping key has an ACL which permits the calling user to use that key with the wrap operation. In our model, this restriction is enforced by the second parameter in the key type tuple.

3. Every user who has obtained the cryptographic value of the wrapping key is permitted to obtain the cryptographic value of every key which is dependent upon the key being wrapped. Our system requires that l_3 dominate l_2, and by the security lattice definition, a dominating security level comprises a subset of the readers in the dominated level. Thus, even when all users who are permitted to obtain the cryptographic value of the key do so, this check will pass.

4. The wrapping key is not a dependent of the key being wrapped. This check is used to prevent cycles in chains of key wrappings, thereby allowing for the use of encryption schemes that may be insecure in the presence of cyclic chains. This property is not dealt with by our type system, requiring either explicit checks (as Cachin and Chandran do) or the use of an encryption scheme that remains secure when key encryption cycles exist.[2] For the purposes of this analysis, we assume the latter, therefore this check is redundant.

5. The key being wrapped is extractable. The type of the result requires that the security level of the wrapped key is dominated by $(\mathsf{x}, \mathcal{R})$ and this can only occur for extractable keys.

6. The key being wrapped is not the public half of a public-private key pair. As we only consider symmetric keys in this model, this check will pass.

7. The wrapping key has only ever been used in the wrap or unwrap operations. The security level of the operand for the wrapping key is specified as $(\mathsf{x}, \mathsf{k}, \mathcal{R}')$ therefore it follows from the forms of types for keys in this model that only key-encrypting keys can be given this type. As a result, this check will pass.

Since all relevant checks will pass, our type system provides a similar guarantee as Cachin and Chandran for the security of well-typed programs which make use of this API function.

[2] It is possible to modify our type system to require the security level of keys to be greater than the security level of the data on which they operate without affecting the security result.

unwrap(k,wk,r)

l
$\langle \mathsf{bool}, l \rangle$
unwrap($hKey : \langle \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle\ \mathsf{loc}, (\mathsf{x}, \mathsf{d}, \{u\}) \rangle$,
$wKey : \langle \mathsf{enc}(\langle l_1'\ l_2'\ \mathsf{key}, (\mathsf{x}, \mathsf{k}, \emptyset) \rangle), (\mathsf{x}, \mathsf{d}, \{u\}) \rangle$,
$pRes : \langle \langle l_1'\ l_2'\ \mathsf{key}, (\mathsf{x}, \mathsf{k}, \mathcal{R}) \rangle\ \mathsf{loc}, (\mathsf{x}, \mathsf{d}, \{u\}) \rangle$) {
try key = sdec(*$hKey$, $wKey$) in
if (*$pRes$ == key) then
TRUE
else
if (*$pRes$ == 0) then
let $x = (pRes := key)$ in TRUE
else
FALSE
else
FALSE
}

Cachin and Chandran specify four checks that must pass before the unwrap operation will attempt to import the encrypted key:

1. The type of the wrapping key is *symmetric*. As noted earlier, our model only considers symmetric keys, therefore this check will pass.

2. The wrapping key has an ACL which permits the calling user to use that key with the unwrap operation. In our model, this restriction is enforced by the second parameter in the key type tuple.

3. No user will have been able to obtain the cryptographic value of the wrapping key, to prevent the importing of known keys. This restriction is enforced in our type system by requiring the set of readers for the unwrapping key to be empty, therefore this check will pass.

4. The unwrapping key has only ever been used in the wrap or unwrap operations. The security level of the operand for the unwrapping key is specified as $(\mathsf{x}, \mathsf{k}, \mathcal{R})$ therefore it follows from the forms of types for keys in this model that only key-encrypting keys can be given this type. As a result, this check will pass.

When importing a symmetric key, Cachin and Chandran require that all users which are permitted to obtain the cryptographic value of the imported key are also permitted to obtain the cryptographic value of any dependent keys. This follows via our type system as a direct consequence of the security level of keys dominating the security level of their operands — every value that a key encrypts is guaranteed to be readable by those users who can read the encrypting key. Additionally, if a key is already associated with the specified key handle then that key must equal the key being imported.

Finally, the imported key must be extractable, since it has come from outside of the API boundary. This requirement is enforced via our type system by requiring that the type of the imported key is dominated by $(\mathsf{x}, \mathsf{k}, \mathcal{R})$.

There are two minor differences between the above implementation of the unwrap function and the description given by Cachin and Chandran:

1. The above implementation allows the caller to specify the key handle for the imported key, rather than it being freely chosen by the device. This is due to our system not having support for runtime generation of location references, and does not affect the security result we are trying to show.

2. Cachin and Chandran cryptographically associate the attributes of a key with that key's wrapped form, to ensure that they cannot be tampered with during a wrap/unwrap sequence.

5.3 Security Guarantees

In Chapter 4, we proved via Theorem 4.20 that well-typed programs are non-interferent. It follows from this result that any well-typed user program which utilises the above API function implementations cannot learn anything about data items which have a higher security level than that of the user themselves. Based on this observation, we now show that the security properties required by Cachin and Chandran follow from those enforced by our type system:

1. *A user may only obtain the cryptographic value of a key whose ACL permits that user to read the key.*

   By inspection of the type signature for the read function, the key whose value is to be read must have a type that is dominated by $(\mathsf{x}, \mathsf{k}, \{u\})$ — the security level associated with user u. It therefore follows from the subtyping rules and the

lattice ordering rules that all keys which meet this requirement have a type which permits user u to obtain the key's value. Consequently, no user can directly read the value of a key whose ACL does not permit that user to do so.

Additionally, it is necessary to prevent a user from obtaining the value of any key which can encrypt a key that that user is not permitted to obtain. Here, our type system requires that the security level of such a key must dominate that of the key being encrypted. Consequently, if the user is not permitted to obtain the cryptographic value of the key being encrypted, the wrapping key will have a type that prevents them from reading its value.

2. *A user must not be able to distinguish between two pieces of ciphertext (either directly or via API calls) when that user is not permitted to obtain the plaintexts within.*

   This follows directly from the non-interference property that our type system guarantees is possessed by well-typed programs.

3. *An unextractable key must remain inside the cryptographic device and therefore cannot be read or wrapped.*

   By inspection of the type signature for the read function, the key whose value is to be read must have a type that is dominated by $(\mathsf{x}, \mathsf{k}, \{u\})$. Similarly, for the wrap function, the key whose value is to be encrypted must have a type that is dominated by $(\mathsf{x}, \mathsf{k}, \mathcal{R})$. It follows from the subtyping rules and the lattice definition that no unextractable key will meet these requirements. Also, the type signature for the encrypt function prevents it from being used to encrypt keys.

4. *An unextractable key cannot have its ACL modified.*

   In our model, no key may have its ACL modified, therefore this property is guaranteed to be upheld. However, if this restriction was to be relaxed, we could still provide this guarantee as follows:

   - Require that the type of any key whose ACL is to be modified has a security level which is dominated by $(\mathsf{x}, \mathsf{k}, \{u\})$, so that any program which passes in an unextractable key will be ill-typed.
   - The ACL of a key may only be modified such that the corresponding type dominates the original type. This endures that any security properties which held before the ACL was changed still apply afterwards.

5.4 Discussion

Since all of the functions are guaranteed to succeed when programs that call them are well-typed, any well-typed program which calls these API functions is guaranteed to uphold the security properties proven for our system in the previous chapter. In reality though, security API attacks often work by calling functions with values that have different types or forms to that which the function expects. In those scenarios, it is necessary for the API to verify that each function input is indeed of the expected type — and many of the checks which Cachin and Chandran describe are exactly that. However, inputs like ciphertext cannot easily be checked in this way unless the plaintext should comprise some known structure.

The evaluation semantics defined for our type system relies on decryption failure being detectable — implying that the plaintext always has some known structure. In the API commands that we modelled in this chapter, ciphertext parameters are either decrypted with a key that only operates on public data (in the case of the decrypt function), or correspond to wrapped keys (as in the case of the unwrap function). In the former case, our type system ensures that keys which operate on public data can only ever encrypt public data. As such, the only way that meaningful plaintext will be returned is when the ciphertext is indeed public data encrypted with the specified key. In the latter case, the API is free to determine the form of the plaintext in the wrapped key, such that any decryption failure could be detected. Assuming that the API does indeed do this, the model which we have presented in this chapter shows that the checks enforced by Cachin and Chandran are sufficient to provide the security guarantees afforded by our type system, without having to do any analysis or proofs other than what we have shown in this chapter.

Chapter 6

Conclusions

We began by asking how to guarantee that a security API implementation does not allow sensitive information to be obtained through some unbounded sequence of API function calls. To answer this question, we developed a type system for a simple imperative language with cryptographic primitives, such that any well-typed API was guaranteed to be both *confidentiality-preserving* and *non-interferent*, provided that the use of the API functions is also well-typed.

When analysing security APIs, the adversary has capabilities that are not present in related areas such as security protocols, such as the ability to execute encryption and decryption operations using unknown keys. This particular difference affects non-interference, since it may allow the adversary to distinguish ciphertexts that he cannot decrypt himself.

Our typing rules are independent of the security lattice, allowing for a more faithful representation of the API implementation being modelled. This flexibility also enables usage restrictions to be encoded in the security lattice, as was done in Chapter 5 with our model of Cachin and Chandran's API.

As is typical for type-based static analysis, the computational complexity of using our type system to obtain these security guarantees scales linearly with the size of the API model. Contrast this to the exponential time and/or memory requirements of solutions which track the current knowledge of the adversary (e.g., [Keighren, 2006], [Cortier et al., 2007]).

Given the ubiquity of security APIs in the modern world, and the responsibilities placed upon them, it is critical that their design and implementation is backed by the appropriate formal verification tools. We believe such tools must necessarily employ API-agnostic static analysis techniques such as those presented in this thesis.

6.1 Challenges

We faced a number of challenges during the course of our research:

- The ability of the adversary to execute decryptions without necessarily having knowledge of the key or ciphertext meant that our notion of indistinguishability had to 'look inside' ciphertext even when it was produced with a confidential key. Ensuring that our definitions were both intuitive and accurate took considerable effort. Another effect of the adversary having this capability was that our typing rules had to be more restrictive, while still allowing us to define well-typed APIs that were practical.

- Our security properties were defined with respect to the untyped system, so that they were independent of the typing rules which would enforce them. As a result, even though we were dealing with well-typed expressions, it was still necessary to bridge the gap between the typed and untyped worlds. This was especially problematic for our main non-interference result because the untyped indistinguishability relation assumed that the values being compared were known to the adversary. This was not true when the values had a type whose security level was greater than that of the adversary. In the end, we had to define a more strict, typed version of non-interference and show that, under those circumstances of interest, the original untyped version also held.

- The proof of Theorem 4.21 (Non-Interference for Typed Expressions) was the most difficult to complete. Once again, it was decryption that was causing us trouble. We wanted to make sure that decryption was not overly restricted, and this meant determining whether or not the plaintext enabled any information to legitimately become known by the adversary. For example, if the plaintext was a key capable of decrypting ciphertext comprising public data, then regardless of whether or not the adversary could obtain that key himself, he could discover if the original decryption was successful or not. A resolution to the proof case for decryption only arose after we identified and proved Lemma 4.24 (Typed Indistinguishability of Arbitrary Values). This lemma showed when the outcome of a branching operation (such as decryption) would be hidden from the adversary.

6.2 Future Work

During the course of our research, we have identified a number of opportunities for future work, mainly with respect to further development of the type system.

The most obvious area where the type system could be improved is in the range of operations that are present. For example, adding support of public-key ciphers would increase the number of APIs which could be represented. Type systems for public key cryptography and authentication already exist in the field of security protocols (e.g., [Abadi and Blanchet, 2001], [Gordon and Jeffrey, 2003]).

Following on from this, it would be useful to apply this type system to popular real-world APIs such as PKCS#11, which has already received much attention from the security API analysis community, [Clulow, 2003b], [Tsalapati, 2007], [Delaune et al., 2008], [Centenaro et al., 2012].

Data integrity is another security property which our type system could be extended to handle. Confidentiality and non-interference are concerned only with data flows from higher to lower security levels, whereas integrity considers how low-security data may influence the values of high-security data.

One area which we were unable to address in this thesis is that of untyped adversary code. That is, to show that any sequence of well-typed API calls would maintain the desired security properties, even when the adversary was attempting to subvert the types of function inputs. One approach would be to ensure that the API code enforces the input types, rather than simply assuming that they hold — comparable to secure compilation (e.g., [Fournet and Rezk, 2008]).

On a similar note, our type system does not consider insecure information flows arising from specific memory updates (i.e., when some confidential value controls which one of two publicly accessible locations are updated). As this is a temporal property, any static analysis would have to include a record of which locations could be modified by an expression, and ensure that changes to confidential values did not affect this set.

Finally, it would be beneficial to have an implementation of the type system that enabled an API design to be automatically type-checked, either using a stand-alone verifier, or interactively in an editor similar to what exists for traditional programming languages in IDEs such as Eclipse.

Bibliography

M. Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, Sept. 1999.

M. Abadi and B. Blanchet. Secrecy Types for Asymmetric Communication. In F. Honsell and M. Miculan, editors, *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, Apr. 2001. of Software (ETAPS 2001).

M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. Technical Report 149, Systems Research Center, Digital Equipment Corporation, Jan. 1998. Available online from `http://www.hpl.hp.com/techreports/`.

M. Abadi and P. Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, pages 3–22. Springer-Verlag, 2000.

R. J. Anderson. The Correctness of Crypto Transaction Sets, 8th International Workshop on Security Protocols, Cambridge, UK, Apr. 2000. Transcript available online at `http://www.cl.cam.ac.uk/ftp/users/rja14/protocols00.pdf`.

A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-Masked Flows. *Journal of Theoretical Computer Science*, 402(2-3):82–101, Aug. 2008.

M. Backes and B. Pfitzmann. Computational Probabilistic Non-Interference. In D. Gollmann, G. Karjoth, and M. Waidner, editors, *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS 2002)*, volume 2502 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, Oct. 2002.

D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report MTR-2547, Vol. I, The MITRE Corporation, Bedford, MA, Mar. 1973.

D. E. Bell and L. J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical Report MTR-2997, The MITRE Corporation, Bedford, MA, Mar. 1976.

M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation. In *38th Annual Symposium on Foundations of Computer Science (FOCS 97)*, pages 394–403. IEEE Computer Society Press, Oct. 1997.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement Types for Secure Implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 17–32. IEEE Computer Society Press, June 2008.

K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, The MITRE Corporation, Bedford, MA, Apr. 1977.

M. K. Bond. A Chosen Key Difference Attack on Control Vectors. Nov. 2000. Available from `http://www.cl.cam.ac.uk/~{}mkb23/research.html`.

M. K. Bond. *Understanding Security APIs*. PhD thesis, University of Cambridge, Jan. 2004.

L. Buttyán and T. V. Thong. Security API Analysis with the Spi Calculus. *Híradástechnika*, Jan. 2008.

C. Cachin and N. Chandran. A Secure Cryptographic Token Interface. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 141–153. IEEE Computer Society Press, 2009.

M. Centenaro, R. Focardi, F. L. Luccio, and G. Steel. Type-based Analysis of PIN Processing APIs. In M. Backes and P. Ning, editors, *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS '09)*, number 5789 in Lecture Notes in Computer Science, pages 53–68. Springer-Verlag, Sept. 2009.

M. Centenaro, R. Focardi, and F. L. Luccio. Type-based Analysis of PKCS#11 Key Management. In P. Degano and J. D. Guttman, editors, *Proceedings of the First International Conference on Principles of Security and Trust*, number 7215 in Lecture Notes in Computer Science, pages 349–368. Springer-Verlag, Mar. 2012.

A. Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

J. Clulow. The Design and Analysis of Cryptographic APIs for Security Devices. Master's thesis, University of Natal, Durban, Jan. 2003.

J. S. Clulow. On the Security of PKCS #11. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Proceedings of the Fifth International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *Lecture Notes in Computer Science*, pages 411–425, Cologne, Germany, Sept. 2003. Springer-Verlag.

V. Cortier, G. Keighren, and G. Steel. Automatic Analysis of the Security of XOR-Based Key Management Schemes. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, number 4424 in Lecture Notes in Computer Science, pages 538–552. Springer-Verlag, Mar. 2007.

S. Delaune, S. Kremer, and G. Steel. Formal Analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 331–344. IEEE Computer Society Press, June 2008.

D. E. Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, May 1975.

D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

J. S. Fenton. *Information Protection Systems*. PhD thesis, University of Cambridge, 1974.

R. Focardi, F. L. Luccio, and G. Steel. An Introduction to Security API Analysis. In A. Aldini and R. Gorrieri, editors, *Foundations of Security Analysis and Design VI — FOSAD Tutorial Lectures*, volume 6858 of *Lecture Notes in Computer Science*, pages 35–65. Springer-Verlag, Sept. 2011.

C. Fournet and T. Rezk. Cryptographically Sound Implementations for Typed Information-Flow Security. In *Proceedings of the* 35th *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*, pages 323–335.

J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, Apr. 1982.

A. D. Gordon and A. Jeffrey. Authenticity by Typing for Security Protocols. *Journal of Computer Security*, 11(4):451–520, 2003.

G. Keighren. Model Checking Security APIs. Master's thesis, School of Informatics, University of Edinburgh, Aug. 2006. Available online at `http://www.inf.ed.ac.uk/publications/thesis/online/IM060368.pdf`.

P. Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In D. Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP 2001)*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, Apr. 2001.

P. Laud. On the Computational Soundness of Cryptographically-Masked Flows. In *Proceedings of the* 35th *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*, pages 337–348. ACM Press, Jan. 2008.

D. McCullough. Noninterference and the Composability of Security Properties. *IEEE Symposium on Security and Privacy*, 0:177–186, May 1988.

R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, pt I. *Information and Computation*, 100(1):1–40, Sept. 1992.

A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing Robust Declassification and Qualified Robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

O. Pereira and J.-J. Quisquater. On the Perfect Encryption Assumption. In P. Degano, editor, *Proceedings of the Workshop on Issues in the Theory of Security (WITS 2000)*, pages 42–45, July 2000.

A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

R. E. Smith. *Multilevel Security*, volume 3, chapter 205, pages 972–986. John Wiley & Sons, 3rd edition, 2006. Also available online at `http://www.cryptosmith.com/multilevel/intro`.

E. Sumii and B. C. Pierce. Logical Relations for Encryption. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, pages 256–269. IEEE Computer Society Press, June 2001.

E. Sumii and B. C. Pierce. Logical Relations for Encryption. *Journal of Computer Security*, 11(4):521–554, 2003.

E. Tsalapati. Analysis of PKCS #11 using AVISPA Tools. Master's thesis, School of Informatics, University of Edinburgh, Aug. 2007.

D. M. Volpano and G. Smith. A Type-Based Approach to Program Security. In *Proceedings of TAPSOFT'97, Colloqium on Formal Approaches in Software Engineering*, pages 607–621, 1997.

D. M. Volpano, G. Smith, and C. E. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

P. Youn. The Analysis of Cryptographic APIs using the Theorem Prover Otter. Master's thesis, Massachusetts Institute of Technology, May 2004.

CCA Basic Services Reference and Guide, Release 3.30. *CCA Basic Services Reference and Guide, Release 3.30*, Sept. 2008. Available online at `http://www-03.ibm.com/security/cryptocards/pdfs/bs330.pdf`.

HMG Security Policy Framework. HMG Security Policy Framework, v1.0, Dec. 2008. Available online at `http://www.cabinetoffice.gov.uk/media/cabinetoffice/corp/assets/foi/classifications.pdf`.

PKCS #11: Cryptographic Token Interface Standard. PKCS #11: Cryptographic Token Interface Standard. `http://www.rsa.com/rsalabs/node.asp?id=2133`.

Appendix A

Proofs

This appendix contains the full proofs of the lemmas and theorems which were omitted from the main body of the Thesis. In the proofs which follow, underlined statements correspond to those results that must hold for the theorem or lemma to be true. Recall that the security properties only hold for *terminating* expressions.

A.1 Confidentiality

Here, we present the definitions and full proofs of the lemmas and theorems which are required for the proof of Theorem 4.17 (Confidentiality Preservation).

A.1.1 Lemmas

Lemma A.1 (Inversion of the Subtype Relation)**.**
If $T <: \langle E, l \rangle$ *then* $T \equiv \langle E, l' \rangle$ *where* $l' \leq l$.

Proof. By induction on the structure of the derivation of $\mathsf{T} <: \mathsf{T}'$ By inspection of the subtyping rules, there are two possible cases for the rule at the root of the derivation:

- Case S-ST:

  Here, $\mathsf{T} = \langle \mathsf{E}, l' \rangle$, where $l' \leq l$, therefore the result is immediate.

- Case S-REFL:

  Here, $\mathsf{T} = \mathsf{T}' = \langle \mathsf{E}, l \rangle$. Since $\mathsf{E} <: \mathsf{E}$ follows via S-REFL and $l \leq l$ is true, the result holds.

Lemma A.2 (Inversion of the Typing Relation for Values)**.**

1. *If* $\vdash_{l_w} c : T$ *w.r.t.* Σ *then* $T = \langle \mathsf{data}, l' \rangle$*, where* $c : \langle \mathsf{data}, l \rangle \in \Sigma$ *and* $(l \curlyvee l_w) \leq l'$
2. *If* $\vdash_{l_w} a : T$ *w.r.t.* Σ *then* $T = \langle S\,\mathsf{loc}, l' \rangle$*, where* $a : \langle S\,\mathsf{loc}, l \rangle \in \Sigma$*,* $l_w \leq \lfloor S \rfloor$ *and* $(l \curlyvee l_w) \leq l'$
3. *If* $\vdash_{l_w} k : T$ *w.r.t.* Σ *then* $T = \langle l_1\, l_2\, \mathsf{key}, l \rangle$*, where* $k : \langle l_1\, l_2\, \mathsf{key}, l_3 \rangle \in \Sigma$ *and* $l_w \leq l_1 \leq l_2 \leq l_3 \leq l$
4. *If* $\vdash_{l_w} ctxt(n, k, v) : T$ *w.r.t.* Σ *then* $\vdash_{l_w} k : \langle \lfloor \langle E, l \rangle \rfloor\, l\, \mathsf{key}, \top \rangle$ *w.r.t.* Σ*,* $\vdash_{l_w} v : \langle E, l \rangle$ *w.r.t.* Σ *and* $T = \langle \mathsf{enc}(\langle E, l \rangle), l' \rangle$
5. *If* $\vdash_{l_w} \text{TRUE} : T$ *then* $T = \langle \mathsf{bool}, l \rangle$*, where* $l_w \leq l$
6. *If* $\vdash_{l_w} \text{FALSE} : T$ *then* $T = \langle \mathsf{bool}, l \rangle$*, where* $l_w \leq l$

Proof. By induction on the structure of the derivation of $\vdash_{l_w} v : \mathsf{T}$

1. Here $v = c$ therefore, by inspection of the typing rules, either T-CNST or T-SUB must be at the root of the derivation.
   - Case T-CNST:

     Here, we have $\mathsf{T} = \langle \mathsf{data}, l \curlyvee l_w \rangle$ and $c : \langle \mathsf{data}, l \rangle \in \Sigma$ therefore the result is immediate, since $(l \curlyvee l_w) \leq (l \curlyvee l_w)$.
   - Case T-SUB:

     We have:

     $$\vdash_{l'_w} c : \mathsf{S}'$$
     $$l_w \leq l'_w$$
     $$\mathsf{S}' <: \mathsf{T}$$

     By the induction hypothesis:

     $$\mathsf{S}' = \langle \mathsf{data}, l'' \rangle$$
     $$\underline{c : \langle \mathsf{data}, l \rangle \in \Sigma}$$
     $$(l \curlyvee l'_w) \;\leq\; l''$$

     By Lemma A.1 (Inversion of the Subtype Relation):

     $$\underline{\mathsf{T} = \langle \mathsf{data}, l' \rangle}$$
     $$l'' \leq l'$$

     By the properties of $\curlyvee$:

     $$(l \curlyvee l_w) \;\leq\; (l \curlyvee l'_w) \quad \Rightarrow \quad \underline{(l \curlyvee l_w) \;\leq\; l'}$$

2. Here $v = a$ therefore, by inspection of the typing rules, either T-LOC or T-SUB must be at the root of the derivation.

   - Case T-LOC:

     Here, we have $\mathsf{T} = \langle \mathsf{S\ loc}, l \curlyvee l_w \rangle$, $l_w \leq \lfloor \mathsf{S} \rfloor$ and $a : \langle \mathsf{data}, l \rangle \in \Sigma$ therefore the result is immediate, since $(l \curlyvee l_w) \leq (l \curlyvee l_w)$.

   - Case T-SUB:

     We have:

     $$\vdash_{l'_w} a : \mathsf{S}''$$
     $$l_w \leq l'_w$$
     $$\mathsf{S}' <: \mathsf{T}$$

     By the induction hypothesis:

     $$\mathsf{S}' = \langle \mathsf{S\ loc}, l'' \rangle$$
     $$l'_w \leq \lfloor \mathsf{S} \rfloor \quad \Rightarrow \quad \underline{l_w \leq \lfloor \mathsf{S} \rfloor}$$
     $$\underline{a : \langle \mathsf{S\ loc}, l \rangle \in \Sigma}$$
     $$(l \curlyvee l'_w) \ \leq \ l''$$

     By Lemma A.1 (Inversion of the Subtype Relation):

     $$\underline{\mathsf{T} = \langle \mathsf{S\ loc}, l' \rangle}$$
     $$l'' \leq l'$$

     By the properties of $\curlyvee$:

     $$(l \curlyvee l_w) \ \leq \ (l \curlyvee l'_w) \quad \Rightarrow \quad \underline{(l \curlyvee l_w) \ \leq \ l'}$$

3. Here $v = k$ therefore, by inspection of the typing rules, either T-KEY or T-SUB must be at the root of the derivation.

   - Case T-KEY:

     Here, we have $\mathsf{T} = \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle$, $k : \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle \in \Sigma$, $l_1 \leq l_2 \leq l_3$ and $l_w = l_1$, therefore the result is immediate.

   - Case T-SUB:

     We have:

     $$\vdash_{l'_w} k : \mathsf{S}'$$
     $$l_w \leq l'_w$$
     $$\mathsf{S}' <: \mathsf{T}$$

By the induction hypothesis:

$$\mathsf{S}' = \langle l_1\ l_2\ \mathsf{key}, l' \rangle$$

$$\underline{k : \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle \in \Sigma}$$

$$l'_w \leq l_1 \leq l_2 \leq l_3 \leq l' \quad \Rightarrow \quad l_w \leq l_1 \leq l_2 \leq l_3 \leq l'$$

By Lemma A.1 (Inversion of the Subtype Relation):

$$\underline{\mathsf{T} = \langle l_1\ l_2\ \mathsf{key}, l \rangle}$$

$$l' \leq l \quad \Rightarrow \quad \underline{l_w \leq l_1 \leq l_2 \leq l_3 \leq l}$$

4. Here, $v = \mathrm{ctxt}(n, k, v_m)$ therefore, by inspection of the typing rules, either T-CTXT or T-SUB must be at the root of the derivation.

   - Case T-CTXT:

     Here, we have $\mathsf{T} = \langle \mathsf{enc}(\langle \mathsf{E}, l \rangle), \bot \rangle$, $\vdash_{l_w} k : \langle \lfloor \langle \mathsf{E}, l \rangle \rfloor\ l\ \mathsf{key}, \top \rangle$ w.r.t. Σ and $\vdash_{l_w} v_m : \langle \mathsf{E}, l \rangle$ w.r.t. Σ, therefore the result is immediate.

   - Case T-SUB:

     We have:

     $$\vdash_{l'_w} \mathrm{ctxt}(n, k, v_m) : \mathsf{S}' \text{ w.r.t. } \Sigma$$

     $$\mathsf{S}' <: \mathsf{T}$$

     By the induction hypothesis:

     $$\mathsf{S}' = \langle \mathsf{enc}(\langle \mathsf{E}, l \rangle), l'' \rangle$$

     $$\vdash_{l'_w} k : \langle \lfloor \langle \mathsf{E}, l \rangle \rfloor\ l\ \mathsf{key}, \top \rangle \text{ w.r.t. } \Sigma$$

     $$\vdash_{l'_w} v_m : \langle \mathsf{E}, l \rangle \text{ w.r.t. } \Sigma$$

     By Lemma A.1 (Inversion of the Subtype Relation):

     $$\underline{\mathsf{T} = \langle \mathsf{enc}(\langle \mathsf{E}, l \rangle), l' \rangle}$$

     By T-SUB and S-REFL:

     $$\underline{\vdash_{l_w} k : \langle \lfloor \langle \mathsf{E}, l \rangle \rfloor\ l\ \mathsf{key}, \top \rangle \text{ w.r.t. } \Sigma}$$

     $$\underline{\vdash_{l_w} v_m : \langle \mathsf{E}, l \rangle \text{ w.r.t. } \Sigma}$$

5. Here, $v = \mathrm{TRUE}$ therefore, by inspection of the typing rules, either T-TRUE or T-SUB must be at the root of the derivation.

   - Case T-TRUE:

     Here, we have $\mathsf{T} = \langle \mathsf{bool}, l_w \rangle$ therefore the result is immediate.

- Case T-SUB:

  We have:

  $$\vdash_{l'_w} \text{TRUE} : \mathsf{S}'$$

  $$\mathsf{S}' <: \mathsf{T}$$

  $$l_w \leq l'_w$$

  By the induction hypothesis:

  $$\mathsf{S}' = \langle \mathsf{bool}, l' \rangle$$

  $$l'_w \leq l' \quad \Rightarrow \quad l_w \leq l'$$

  By Lemma A.1 (Inversion of the Subtype Relation):

  $$\underline{\mathsf{T} = \langle \mathsf{bool}, l \rangle}$$

  $$l' \leq l \quad \Rightarrow \quad \underline{l_w \leq l}$$

6. Here, v = FALSE therefore, by inspection of the typing rules, either T-FALSE or T-SUB must be at the root of the derivation.

   - Case T-FALSE:

     Here, we have $\mathsf{T} = \langle \mathsf{bool}, l_w \rangle$ therefore the result is immediate.

   - Case T-SUB:

     We have:

     $$\vdash_{l'_w} \text{FALSE} : \mathsf{S}'$$

     $$\mathsf{S}' <: \mathsf{T}$$

     $$l_w \leq l'_w$$

     By the induction hypothesis:

     $$\mathsf{S}' = \langle \mathsf{bool}, l' \rangle$$

     $$l'_w \leq l' \quad \Rightarrow \quad l_w \leq l'$$

     By Lemma A.1 (Inversion of the Subtype Relation):

     $$\underline{\mathsf{T} = \langle \mathsf{bool}, l \rangle}$$

     $$l' \leq l \quad \Rightarrow \quad \underline{l_w \leq l}$$

□

Lemma A.3 (Canonical Forms)**.**

1. *If* $\vdash_{l_w} v : \langle \mathsf{data}, l \rangle$ *w.r.t.* Σ *then v is of the form c*
2. *If* $\vdash_{l_w} v : \langle \mathsf{bool}, l \rangle$ *w.r.t.* Σ *then v is of the form* TRUE *or* FALSE
3. *If* $\vdash_{l_w} v : \langle \mathsf{S\,loc}, l \rangle$ *w.r.t.* Σ *then v is of the form a*
4. *If* $\vdash_{l_w} v : \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle$ *w.r.t.* Σ *then v is of the form k*
5. *If* $\vdash_{l_w} v : \langle \mathsf{enc(S)}, l \rangle$ *w.r.t.* Σ *then v is of the form ctxt(n, k,* v_m*)*

Proof. By induction on the structure of the derivation of $\vdash_{l_w} v : \mathsf{S}$ w.r.t. Σ.

1. Here $\mathsf{S} = \langle \mathsf{data}, l \rangle$ therefore, by inspection of the typing rules, either T-CNST or T-SUB must be at the root of the derivation. In the former case, we have $v = c$ thus the result is immediate. In the latter case, we have $\vdash_{l'_w} v : \mathsf{S}'$ w.r.t. Σ, where $\mathsf{S}' <: \langle \mathsf{data}, l \rangle$. It follows from Lemma A.1 (Inversion of the Subtyping Relation) that $\mathsf{S}' \equiv \langle \mathsf{data}, l' \rangle$ therefore the result follows from the induction hypothesis.

2. Here $\mathsf{S} = \langle \mathsf{bool}, l \rangle$ therefore, by inspection of the typing rules, T-TRUE, T-FALSE or T-SUB must be at the root of the derivation. In the first two cases, we have $v =$ TRUE and $v =$ FALSE respectively, thus the result is immediate. In the third case, we have $\vdash_{l'_w} v : \mathsf{S}'$ w.r.t. Σ, where $\mathsf{S}' <: \langle \mathsf{bool}, l \rangle$. It follows from Lemma A.1 (Inversion of the Subtyping Relation) that $\mathsf{S}' \equiv \langle \mathsf{bool}, l' \rangle$ therefore the result follows from the induction hypothesis.

3. Here $\mathsf{S} = \langle \mathsf{S\,loc}, l \rangle$ therefore, by inspection of the typing rules, either T-LOC or T-SUB must be at the root of the derivation. In the former case, we have $v = a$ thus the result is immediate. In the latter case, we have $\vdash_{l'_w} v : \mathsf{S}'$ w.r.t. Σ, where $\mathsf{S}' <: \langle \mathsf{S\,loc}, l \rangle$. It follows from Lemma A.1 (Inversion of the Subtyping Relation) that $\mathsf{S}' \equiv \langle \mathsf{S\,loc}, l' \rangle$ therefore the result follows from the induction hypothesis.

4. Here $\mathsf{S} = \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle$ therefore, by inspection of the typing rules, either T-KEY or T-SUB must be at the root of the derivation. In the former case, we have $v = k$ thus the result is immediate. In the latter case, we have $\vdash_{l'_w} v : \mathsf{S}'$ w.r.t. Σ, where $\mathsf{S}' <: \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle$. It follows from Lemma A.1 (Inversion of the Subtyping Relation) that $\mathsf{S}' \equiv \langle l_1\ l_2\ \mathsf{key}, l'_3 \rangle$ therefore the result follows from the induction hypothesis.

5. Here $\mathsf{S} = \langle \mathsf{enc}(\mathsf{S}_m), l \rangle$ therefore, by inspection of the typing rules, either T-CTXT or T-SUB must be at the root of the derivation. In the former case, we have $v =$ ctxt(n, k, v_m) thus the result is immediate. In the latter case, we have $\vdash_{l'_w} v : \mathsf{S}'$

w.r.t. Σ, where $\mathsf{S}' <: \langle \mathsf{enc}(\mathsf{S}_m), l \rangle$. It follows from Lemma A.1 (Inversion of the Subtyping Relation) that $\mathsf{S}' \equiv \langle \mathsf{enc}(\mathsf{S}_m), l' \rangle$ therefore the result follows from the induction hypothesis. □

Lemma A.4 (Sufficient Minimum Write Level)**.**

If $\vdash_{l_w} v : S$ *w.r.t.* Σ *then* $\vdash_{\lfloor S \rfloor} v : S$ *w.r.t.* Σ.

Proof.

By induction on the form of S.

- Case $\langle \mathsf{data}, l \rangle$:

  By Lemma A.3 (Canonical Forms):

  $$v = c$$

  By Lemma A.2 (Inversion of the Typing Relation):

  $$c : \langle \mathsf{data}, l' \rangle \in \Sigma$$

  $$(l' \curlyvee l_w) \leq l$$

  By the properties of $\curlyvee$:

  $$l' \leq l \quad \Rightarrow \quad (l' \curlyvee l) = l$$

  By Definition 4.13 (Minimum Level of a Security Type):

  $$\lfloor \langle \mathsf{data}, l \rangle \rfloor = l$$

  By T-CNST:

  $$\vdash_l c : \langle \mathsf{data}, l \rangle \qquad \equiv \qquad \underline{\vdash_{\lfloor \mathsf{S} \rfloor} v : \mathsf{S}}$$

- Case $\langle \mathsf{bool}, l \rangle$:

  By Lemma A.3 (Canonical Forms):

  $$v \in \{\text{TRUE}, \text{FALSE}\}$$

  By Definition 4.13 (Minimum Level of a Security Type):

  $$\lfloor \langle \mathsf{bool}, l \rangle \rfloor = l$$

  By T-TRUE or T-FALSE:

  $$\vdash_l v : \langle \mathsf{bool}, l \rangle \qquad \equiv \qquad \underline{\vdash_{\lfloor \mathsf{S} \rfloor} v : \mathsf{S}}$$

- Case $\langle \mathsf{S}' \, \mathsf{loc}, l \rangle$:

  By Lemma A.3 (Canonical Forms):

  $$v = a$$

By Lemma A.2 (Inversion of the Typing Relation):

$a : \langle \mathsf{S}' \ \mathsf{loc}, l' \rangle \in \Sigma$

$(l' \curlyvee l_w) \leq l \qquad \Rightarrow \qquad l' \leq l$

By T-LOC:

$\vdash_{\lfloor \mathsf{S}' \rfloor} a : \langle \mathsf{S}' \ \mathsf{loc}, (l' \curlyvee \lfloor \mathsf{S}' \rfloor) \rangle$

By the properties of $\curlyvee$:

$(l' \curlyvee \lfloor \mathsf{S}' \rfloor) \leq l' \qquad \Rightarrow \qquad (l' \curlyvee \lfloor \mathsf{S}' \rfloor) \leq l$

By Definition 4.13 (Minimum Level of a Security Type):

$\lfloor \langle \mathsf{S}' \ \mathsf{loc}, l \rangle \rfloor = \lfloor \mathsf{S}' \rfloor \curlywedge l \qquad \Rightarrow \qquad \lfloor \mathsf{S} \rfloor \leq \lfloor \mathsf{S}' \rfloor$

By T-SUB, S-ST and S-REFL:

$\vdash_{\lfloor \mathsf{S} \rfloor} a : \langle \mathsf{S}' \ \mathsf{loc}, l \rangle \qquad \equiv \qquad \underline{\vdash_{\lfloor \mathsf{S} \rfloor} v : \mathsf{S}}$

- Case $\langle l_1 \ l_2 \ \mathsf{key}, l_3 \rangle$:

  By Lemma A.3 (Canonical Forms):

  $v = k$

  By Lemma A.2 (Inversion of the Typing Relation):

  $k : \langle l_1 \ l_2 \ \mathsf{key}, l_3' \rangle \in \Sigma$

  $l_3' \leq l_3$

  By Definition 4.13 (Minimum Level of a Security Type):

  $\lfloor \langle l_1 \ l_2 \ \mathsf{key}, l_3 \rangle \rfloor = l_1$

  By T-KEY:

  $\vdash_{l_1} k : \langle l_1 \ l_2 \ \mathsf{key}, l_3' \rangle$

  By T-SUB, S-ST and S-REFL:

  $\vdash_{l_1} k : \langle l_1 \ l_2 \ \mathsf{key}, l_3 \rangle \qquad \equiv \qquad \underline{\vdash_{\lfloor \mathsf{S} \rfloor} v : \mathsf{S}}$

- Case $\langle \mathsf{enc}(\mathsf{S}_m), l \rangle$:

  By Lemma A.3 (Canonical Forms):

  $v = \mathrm{ctxt}(n, k, v_m)$

By Lemma A.2 (Inversion of the Typing Relation):

$$\left.\begin{array}{l} \vdash_{l_w} k : \langle \lfloor \mathsf{S}_m \rfloor \; l_m \; \mathsf{key}, \top \rangle \\ \vdash_{l_w} v_m : \mathsf{S}_m \\ l_w \leq \lfloor \mathsf{S}_m \rfloor \leq l_m \end{array}\right\} \text{ where } \mathsf{S}_m = \langle \mathsf{E}_m, l_m \rangle$$

By Definition 4.13 (Minimum Level of a Security Type):

$$\lfloor \langle \lfloor \mathsf{S}_m \rfloor \; l_m \; \mathsf{key}, \top \rangle \rfloor = \lfloor \mathsf{S}_m \rfloor$$

$$\lfloor \langle \mathsf{enc}(\mathsf{S}_m), l \rangle \rfloor = (\lfloor \mathsf{S}_m \rfloor \curlywedge l) \quad \Rightarrow \quad \lfloor \mathsf{S} \rfloor \leq \lfloor \mathsf{S}_m \rfloor$$

By the induction hypothesis:

$$\vdash_{\lfloor \mathsf{S}_m \rfloor} k : \langle \lfloor \mathsf{S}_m \rfloor \; l_m \; \mathsf{key}, \top \rangle$$

$$\vdash_{\lfloor \mathsf{S}_m \rfloor} v_m : \mathsf{S}_m$$

By T-SUB and S-REFL:

$$\vdash_{\lfloor \mathsf{S} \rfloor} k : \langle \lfloor \mathsf{S}_m \rfloor \; l_m \; \mathsf{key}, \top \rangle$$

$$\vdash_{\lfloor \mathsf{S} \rfloor} v_m : \mathsf{S}_m$$

By T-CTXT:

$$\vdash_{\lfloor \mathsf{S} \rfloor} \mathrm{ctxt}(n, k, v_m) : \langle \mathsf{enc}(\mathsf{S}_m), \bot \rangle$$

By T-SUB and S-ST and S-REFL:

$$\vdash_{\lfloor \mathsf{S} \rfloor} \mathrm{ctxt}(n, k, v_m) : \langle \mathsf{enc}(\mathsf{S}_m), l \rangle \quad \equiv \quad \underline{\vdash_{\lfloor \mathsf{S} \rfloor} v : \mathsf{S}} \qquad \square$$

Lemma A.5 (Lower Bound on Security Level of Non-Ciphertext Types)**.**
If $\vdash_{l_w} v : \langle E, l \rangle$ *w.r.t.* Σ, $E \not\equiv enc(S)$ *and* Σ *is valid then* $l_w \leq l$.

Proof. By induction on the structure of the derivation of $\vdash_{l_w} v : \langle \mathsf{E}, l \rangle$ *w.r.t.* Σ.

- Cases T-CNST and T-LOC:

  Here, $l = (l \curlyvee l_w)$ therefore the result is immediate.

- Cases T-TRUE and T-FALSE:

  Here, $l = l_w$ therefore the result is immediate.

- Case T-KEY:

  Here, we have $\mathsf{E} = l_1 \; l_2 \; \mathsf{key}$, $l_w = l_1$ and $l_1 \leq l_2 \leq l$, therefore the result follows immediately.

- Case T-CTXT:

  This case cannot occur as we have $\mathsf{E} \not\equiv \mathsf{enc}(\mathsf{S})$ as one of the preconditions.

- Case T-SUB:

  Here, we have $\vdash_{l'_w} v : \mathsf{S}'$ *w.r.t.* Σ, where $l_w \leq l'_w$ and $\mathsf{S}' <: \langle \mathsf{E}, l \rangle$. It follows from Lemma A.1 (Inversion of the Subtyping Relation) that $\mathsf{S}' = \langle \mathsf{E}, l' \rangle$, where $l' \leq l$. For each instance of E that is not of the form $\mathsf{enc}(\mathsf{S})$, it follows by the same lemma that $\mathsf{E}' \not\equiv \mathsf{enc}(\mathsf{S})$. Consequently, the induction hypothesis applies and we get $l'_w \leq l'$, thus the result follows from the transitivity of $\leq$. □

Lemma 4.19 (Upper Bound on Security Level of Well-Typed Values)
For any valid typing domain (Γ, Σ) *and evaluation domain* (Π, Δ) *where* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$*, it follows from* $\vdash_{l_w} v : \langle \mathsf{E}, l \rangle$ *w.r.t.* Σ *that* $lvl_\Delta(v) \leq l$.

Proof. By induction on the structure of the derivation of $\vdash_{l_w} v : \langle \mathsf{E}, l \rangle$ w.r.t. Σ

- Case T-CNST:

  We have:

  $$v = c$$
  $$\langle \mathsf{E}, l \rangle = \langle \mathsf{data}, l' \curlyvee l_w \rangle$$
  $$c : \langle \mathsf{data}, l' \rangle \in \Sigma$$

  By Definition 4.15 (Domain Consistency):

  $$c : l' \in \Delta$$

  By Definition 4.5 (Security Level Function):

  $$lvl_\Delta(c) = l'$$

  By the properties of $\curlyvee$:

  $$l' \leq (l' \curlyvee l_w) \quad \Rightarrow \quad l' \leq l \quad \Rightarrow \quad lvl_\Delta(c) \leq l \quad \equiv \quad \underline{lvl_\Delta(v) \leq l}$$

- Cases T-LOC, T-TRUE and T-FALSE:

  We have:

  $$v = a, \text{TRUE}, \text{FALSE} \qquad \text{(respectively)}$$

  By Definition 4.5 (Security Level Function):

  $$lvl_\Delta(v) = \bot \quad \Rightarrow \quad \underline{lvl_\Delta(v) \leq l}$$

- Case T-KEY:

  We have:

  $$v = k$$
  $$\langle \mathsf{E}, l \rangle = \langle l_1\ l_2\ \mathsf{key}, l \rangle$$
  $$k : \langle l_1\ l_2\ \mathsf{key}, l \rangle \in \Sigma$$

  By Definition 4.15 (Domain Consistency):

  $$k : l \in \Delta$$

  By Definition 4.5 (Security Level Function):

  $$lvl_\Delta(k) = l \quad \Rightarrow \quad lvl_\Delta(k) \leq l \quad \equiv \quad \underline{lvl_\Delta(v) \leq l}$$

- Case T-CTXT:

  We have:

  $$v = \mathrm{ctxt}(n, v_k, v_m)$$
  $$\langle \mathsf{E}, l \rangle = \langle \mathsf{enc}(\langle \mathsf{E}', l' \rangle), \bot \rangle$$
  $$\vdash_{l_w} v_k : \langle l''\ l'\ \mathsf{key}, \top \rangle$$
  $$\vdash_{l_w} v_m : \langle \mathsf{E}', l' \rangle$$

  By Lemma A.3 (Canonical Forms):

  v_k is of the form k

  By Lemma A.2 (Inversion of the Typing Relation for Values):

  $$v_k : \langle l''\ l'\ \mathsf{key}, l_k \rangle \in \Sigma$$
  $$l_w \leq l'' \leq l' \leq l_k$$

  By Definition 4.15 (Domain Consistency):

  $$v_k : l_k \in \Delta$$

  By Definition 4.5 (Security Level Function):

  $$lvl_\Delta(v_k) = l_k \quad \Rightarrow \quad lvl_\Delta(v_k) \geq l'$$

  By application of the induction hypothesis to $\vdash_{l_w} v_m : \langle \mathsf{E}', l' \rangle$:

  $$lvl_\Delta(v_m) \leq l'$$

  By Definition 4.5 (Security Level Function):

  $$lvl_\Delta(\mathrm{ctxt}(n, v_k, v_m)) = \bot \quad \Rightarrow \quad lvl_\Delta(\mathrm{ctxt}(n, v_k, v_m)) \leq l \quad \equiv \quad \underline{lvl_\Delta(v) \leq l}$$

- Case T-SUB:

  We have:

  $$\Gamma \vdash_{l'_w} v : \mathsf{S}'$$
  $$\Gamma \subseteq \emptyset \quad \Rightarrow \quad \Gamma = \emptyset$$
  $$\mathsf{S}' <: \langle \mathsf{E}, l \rangle$$

  By Lemma A.1 (Inversion of the Subtype Relation):

  S' is of the form $\langle \mathsf{E}, l' \rangle$

  $$l' \leq l$$

  By an application of the induction hypothesis:

  $$lvl_\Delta(v) \leq l' \quad \Rightarrow \quad \underline{lvl_\Delta(v) \leq l}$$

  □

A.1.2 Theorems

Theorem 4.18 (Type Safety)

For any valid typing domain (Γ, Σ) *and evaluation domain* (Π, Δ) *where* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$*, if* $\Gamma \vdash_{lw} e : \mathsf{S}$ *w.r.t.* Σ *then either* e *is a value, or there exists some final state* $\langle \Delta', v \rangle$ *such that* $\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta', v \rangle$*,* $(\Gamma, \Sigma) \simeq (\Pi, \Delta')$ *and* $\vdash_{lw} v : \mathsf{S}$ *w.r.t.* Σ*.*

Proof. By induction on the structure of the derivation of $\Gamma \vdash_{lw} e : \mathsf{S}$:

- Cases T-CNST, T-TRUE, T-FALSE, T-LOC, T-KEY and T-CTXT:

  Here, e is a value, therefore the result is immediate.

- Case T-VAR:

  We have:

  $$e = x$$
  $$\mathsf{S} = \langle \mathsf{E}, l \curlyvee l_w \rangle$$
  $$\Gamma = [x : \langle \mathsf{E}, l \rangle]$$
  $$l_w \leq \lfloor \langle \mathsf{E}, l \curlyvee l_w \rangle \rfloor$$

  By Definition 4.15 (Domain Consistency):

  $$(x \mapsto v') \in \Pi$$
  $$\vdash_{\bot} v' : \langle \mathsf{E}, l \rangle \quad \textit{w.r.t.}\ \Sigma$$

  Consequently, E-VAR applies and we get:

  $$\underline{v \;=\; v'}$$
  $$\underline{\Delta' \;=\; \Delta} \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$$

  By the properties of $\curlyvee$:

  $$l \leq (l \curlyvee l_w)$$

  By T-SUB:

  $$\vdash_{\bot} v : \mathsf{S} \quad \textit{w.r.t.}\ \Sigma$$

  By Lemma A.4 (Sufficient Minimum Write Level):

  $$\vdash_{\lfloor \mathsf{S} \rfloor} v : \mathsf{S} \quad \textit{w.r.t.}\ \Sigma$$

  By T-SUB and S-REFL:

  $$\underline{\vdash_{lw} v : \mathsf{S} \ \textit{w.r.t.}\ \Sigma}$$

- Case T-SENC:

  We have:

  $$e = \mathrm{senc}(e_k, e_m)$$
  $$\mathsf{S} = \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \bot \rangle$$
  $$\Gamma \vdash_{l_w} e_k : \langle l'\ l_m\ \mathsf{key}, \top \rangle \ \ \textit{w.r.t.}\ \Sigma$$
  $$\Gamma \vdash_{l_w} e_m : \langle \mathsf{E}_m, l_m \rangle \ \ \textit{w.r.t.}\ \Sigma$$
  $$l' \leq \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor$$

  By inspection of the evaluation rules, only E-SENC may apply, in which case e_k will be evaluated first, followed by e_m. By the induction hypothesis, either e_k is a value, or there exists some state $\langle \Delta'', v_k \rangle$ such that:

  $$\Pi \vdash \langle \Delta, e_k \rangle \Downarrow \langle \Delta'', v_k \rangle$$
  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$
  $$\vdash_{l_w} v_k : \langle l'\ l_m\ \mathsf{key}, \top \rangle \ \ \textit{w.r.t.}\ \Sigma$$

  If e_k is a value, we have:

  $$v_k \ = \ e_k$$
  $$\Delta'' \ = \ \Delta \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

  In both cases, it follows from Lemma A.3 (Canonical Forms) that:

  $$v_k \ = \ k$$

  By a second application of the induction hypothesis, either e_m is a value, or there exists some state $\langle \Delta''', v_m \rangle$ such that:

  $$\Pi \vdash \langle \Delta'', e_m \rangle \Downarrow \langle \Delta''', v_m \rangle$$
  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$
  $$\vdash_{l_w} v_m : \langle \mathsf{E}_m, l_m \rangle \ \ \textit{w.r.t.}\ \Sigma$$

  If e_m is a value, we have:

  $$v_m \ = \ e_m$$
  $$\Delta''' \ = \ \Delta'' \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$

  Consequently, E-SENC does apply and we get:

  $$\underline{v = \mathrm{ctxt}(n, k, v_m)} \quad \textit{for some } n \notin \Delta'''$$
  $$\underline{\Delta' = (\Delta''' \cup n)} \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$$

  By T-CTXT:

  $$\vdash_{l_w} \mathrm{ctxt}(n, k, v_m) : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \bot \rangle \ \ \textit{w.r.t.}\ \Sigma \quad \equiv \quad \underline{\vdash_{l_w} v : \mathsf{S} \ \ \textit{w.r.t.}\ \Sigma}$$

- Case T-SDEC:

  We have:

  $$e \;=\; \text{try sdec}(e_k, e_c) = x \text{ in } e_1 \text{ else } e_2$$

  $$\Gamma \vdash_{l_w} e_k : \langle l'\ l_m\ \mathsf{key}, \top \rangle \textit{ w.r.t. } \Sigma$$

  $$\Gamma \vdash_{l_w} e_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle \textit{ w.r.t. } \Sigma$$

  $$l' \leq \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor$$

  $$x \notin \mathsf{dom}(\Sigma)$$

  $$(\Gamma, x : \langle \mathsf{E}_m, l_m \rangle) \vdash_{(l' \curlyvee l_w)} e_1 : \mathsf{S} \textit{ w.r.t. } \Sigma$$

  $$\Gamma \vdash_{(l' \curlyvee l_w)} e_2 : \mathsf{S} \textit{ w.r.t. } \Sigma$$

  By inspection of the evaluation rules, either E-SDEC1 or E-SDEC2 may apply. In both cases, e_k will be evaluated first, followed by e_c, and then finally by either e_1 or e_2. By the induction hypothesis, either e_k is a value, or there exists some state $\langle \Delta'', v_k \rangle$ such that:

  $$\Pi \vdash \langle \Delta, e_k \rangle \Downarrow \langle \Delta'', v_k \rangle$$

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

  $$\vdash_{l_w} v_k : \langle l'\ l_m\ \mathsf{key}, \top \rangle \;\; \textit{w.r.t. } \Sigma$$

  If e_k is a value, we have:

  $$v_k \;=\; e_k$$

  $$\Delta'' \;=\; \Delta \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

  In both cases, it follows from Lemma A.3 (Canonical Forms) that:

  $$v_k \;=\; k$$

  By a second application of the induction hypothesis, either e_c is a value, or there exists some state $\langle \Delta''', v_c \rangle$ such that:

  $$\Pi \vdash \langle \Delta'', e_c \rangle \Downarrow \langle \Delta''', v_c \rangle$$

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$

  $$\vdash_{l_w} v_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle \;\; \textit{w.r.t. } \Sigma$$

  If e_c is a value, we have:

  $$v_c \;=\; e_c$$

  $$\Delta''' \;=\; \Delta'' \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$

  In both cases, it follows from Lemma A.3 (Canonical Forms) that:

  $$v_c = \text{ctxt}(n, k', v'_m)$$

By Definition 4.15 (Domain Consistency):

$x \notin \mathsf{dom}(\Pi)$

If $k = k'$ then E-SDEC1 applies, otherwise E-SDEC2 applies.

1. E-SDEC1:

   Here, v_c is of the form $\mathrm{ctxt}(n, k, v'_m)$ and e_1 is evaluated next.

   By Lemma A.2 (Inversion of the Typing Relation):

   $\vdash_{l_w} v'_m : \langle \mathsf{E}_m, l_m \rangle$

   By Lemma A.4 (Sufficient Minimum Write Level):

   $\vdash_{\lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor} v'_m : \langle \mathsf{E}_m, l_m \rangle$

   By Definition 4.15 (Domain Consistency):

   $((\Gamma, x : \langle \mathsf{E}_m, l_m \rangle), \Sigma) \simeq (\Pi \cup (x \mapsto v'_m), \Delta''')$

   By the induction hypothesis, either e_1 is a value, or there exists some state $\langle \Delta'''', v' \rangle$ such that:

   $\Pi \cup (x \mapsto v'_m) \vdash \langle \Delta''', e_1 \rangle \Downarrow \langle \Delta'''', v' \rangle$

   $((\Gamma, x : \langle \mathsf{E}_m, l_m \rangle), \Sigma) \simeq (\Pi \cup (x \mapsto v'_m), \Delta'''') \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'''')$

   $\vdash_{(l' \curlyvee l_w)} v' : \mathsf{S} \;\; w.r.t.\; \Sigma$

   If e_1 is a value, we have:

   $v' = e_1$

   $\Delta'''' = \Delta''' \quad \Rightarrow \quad ((\Gamma, x : \langle \mathsf{E}_m, l_m \rangle), \Sigma) \simeq (\Pi \cup (x \mapsto v'_m), \Delta'''')$

   $\Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'''')$

   Consequently, E-SDEC1 does apply and we get:

   $\underline{v = v'} \quad \Rightarrow \quad \vdash_{(l' \curlyvee l_w)} v : \mathsf{S} \; w.r.t.\; \Sigma$

   $\underline{\Delta' = \Delta''''} \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$

   By T-SUB, S-REFL and the properties of $\curlyvee$:

   $\underline{\vdash_{l_w} v : \mathsf{S} \; w.r.t.\; \Sigma}$

2. E-SDEC2:

   Here, v_c is of the form $\mathrm{ctxt}(n, k', v'_m)$, where $k' \neq k$, and e_2 is evaluated next.

   By the induction hypothesis, either e_2 is a value, or there exists some state $\langle \Delta'''', v' \rangle$ such that:

$$\Pi \vdash \langle \Delta'', e_2 \rangle \Downarrow \langle \Delta''', v' \rangle$$

$$(\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$

$$\vdash_{(l' \curlyvee l_w)} v' : \mathsf{S} \textit{ w.r.t. } \Sigma$$

If e_2 is a value, we have:

$$v' = e_2$$

$$\Delta''' = \Delta'' \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$

Consequently, E-SDEC2 does apply and we get:

$$\underline{v = v'} \quad \Rightarrow \quad \vdash_{(l' \curlyvee l_w)} v : \mathsf{S} \textit{ w.r.t. } \Sigma$$

$$\underline{\Delta' = \Delta'''} \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$$

By T-SUB, S-REFL and the properties of $\curlyvee$:

$$\underline{\vdash_{l_w} v : \mathsf{S} \textit{ w.r.t. } \Sigma}$$

- Case T-LET:

  We have:

  $$e = \text{let } x = e_1 \text{ in } e_2$$

  $$\Gamma \vdash_{l_w} e_1 : \mathsf{S}'$$

  $$x \notin \mathsf{dom}(\Gamma)$$

  $$(\Gamma, x : \mathsf{S}') \vdash_{l_w} e_2 : \mathsf{S}$$

  By inspection of the evaluation rules, only E-LET may apply, in which case e_1 will be evaluated first, followed by e_2.

  By the induction hypothesis, either e_1 is a value, or there exists some state $\langle \Delta'', v_1 \rangle$ such that:

  $$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta'', v_1 \rangle$$

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

  $$\Gamma \vdash_{l_w} v_1 : \mathsf{S}' \textit{ w.r.t. } \Sigma$$

  If e_1 is a value, we have:

  $$v_1 = e_1$$

  $$\Delta'' = \Delta \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

  By Lemma A.4 (Sufficient Minimum Write Level):

  $$\vdash_{\lfloor \mathsf{S}' \rfloor} v_1 : \mathsf{S}'$$

By Definition 4.15 (Domain Consistency):

$x \notin \mathsf{dom}(\Pi)$

$((\Gamma, x : \mathsf{S}'), \Sigma) \simeq (\Pi \cup (x \mapsto v_1), \Delta'')$

By a second application of the induction hypothesis, either e_2 is a value, or there exists some state $\langle \Delta''', v_2 \rangle$ such that:

$\Pi \cup (x \mapsto v_1) \vdash \langle \Delta'', e_2 \rangle \Downarrow \langle \Delta''', v_2 \rangle$

$((\Gamma, x : \mathsf{S}'), \Sigma) \simeq (\Pi \cup (x \mapsto v_1), \Delta''')$

$\vdash_{l_w} v_2 : \mathsf{S}$ *w.r.t.* Σ

If e_2 is a value, we have:

$v_2 \;=\; e_2$

$\Delta''' \;=\; \Delta'' \quad \Rightarrow \quad ((\Gamma, x : \mathsf{S}'), \Sigma) \simeq (\Pi \cup (x \mapsto v_1), \Delta''')$

Consequently, E-LET does apply and we get:

$\underline{v = v_2} \quad \Rightarrow \quad \underline{\vdash_{l_w} v : \mathsf{S} \text{ } w.r.t. \text{ } \Sigma}$

$\underline{\Delta' = \Delta'''} \quad \Rightarrow \quad ((\Gamma, x : \mathsf{S}'), \Sigma) \simeq (\Pi \cup (x \mapsto v_1), \Delta') \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$

- Case T-IF:

  We have:

  $e \;=\; \text{if } e_1 \text{ then } e_2 \text{ else } e_3$

  $\Gamma \vdash_{l_w} e_1 : \langle \mathsf{bool}, l \rangle$ *w.r.t.* Σ

  $\Gamma \vdash_{(l_w \curlyvee l)} e_2 : \mathsf{S}$ *w.r.t.* Σ

  $\Gamma \vdash_{(l_w \curlyvee l)} e_3 : \mathsf{S}$ *w.r.t.* Σ

  By inspection of the evaluation rules, either E-IF1 or E-IF2 may apply, where e_1 will be evaluated first, followed by either e_2 or e_3.

  By the induction hypothesis, either e_1 is a value, or there exists some state $\langle \Delta'', v_1 \rangle$ such that:

  $\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta'', v_1 \rangle$

  $(\Gamma, \Sigma) \simeq (\Pi, \Delta'')$

  $\vdash_{l_w} v_1 : \langle \mathsf{bool}, l \rangle$ *w.r.t.* Σ

  If e_1 is a value, we have:

  $v_1 \;=\; e_1$

  $\Delta'' \;=\; \Delta \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'')$

By Lemma A.3 (Canonical Forms):

$v_1 \in \{\text{TRUE}, \text{FALSE}\}$

1. If $v_1 = \text{TRUE}$, only E-IF1 may apply, in which case e_2 is evaluated next.

   By the induction hypothesis, either e_2 is a value, or there exists some state $\langle \Delta'', v_2 \rangle$ such that:

   $\Pi \vdash \langle \Delta', e_2 \rangle \Downarrow \langle \Delta'', v_2 \rangle$

   $(\Gamma, \Sigma) \simeq (\Pi, \Delta'')$

   $\vdash_{(l_w \curlyvee l)} v_2 : \mathsf{S}$ *w.r.t.* Σ

   If e_2 is a value, we have:

   $v_2 = e_2$

   $\Delta'' = \Delta' \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'')$

   Consequently, E-IF1 does apply and we get:

   $\underline{v = v_2} \quad \Rightarrow \quad \vdash_{(l_w \curlyvee l)} v : \mathsf{S}$ *w.r.t.* Σ

   $\underline{\Delta' = \Delta''} \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$

   By T-SUB, S-REFL and the properties of $\curlyvee$:

   $\underline{\vdash_{l_w} v : \mathsf{S}$ *w.r.t.* $\Sigma}$

2. If $v_1 = \text{FALSE}$, only E-IF2 may apply, in which case e_3 is evaluated next.

   By the induction hypothesis, either e_3 is a value, or there exists some state $\langle \Delta'', v_3 \rangle$ such that:

   $\Pi \vdash \langle \Delta', e_3 \rangle \Downarrow \langle \Delta'', v_3 \rangle$

   $(\Gamma, \Sigma) \simeq (\Pi, \Delta'')$

   $\vdash_{(l_w \curlyvee l)} v_3 : \mathsf{S}$ *w.r.t.* Σ

   If e_3 is a value, we have:

   $v_3 = e_3$

   $\Delta'' = \Delta' \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'')$

   Consequently, E-IF2 does apply and we get:

   $\underline{v = v_3} \quad \Rightarrow \quad \vdash_{(l_w \curlyvee l)} v : \mathsf{S}$ *w.r.t.* Σ

   $\underline{\Delta' = \Delta''} \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$

   By T-SUB, S-REFL and the properties of $\curlyvee$:

   $\underline{\vdash_{l_w} v : \mathsf{S}$ *w.r.t.* $\Sigma}$

- Case T-EQ:

  We have:

  $$e \;=\; e_1 == e_2$$

  $$\mathsf{S} \;=\; \langle \mathsf{bool}, l \rangle$$

  $$\Gamma \vdash_{l_w} e_1 : \langle \mathsf{E}, l \rangle \textit{ w.r.t. } \Sigma$$

  $$\Gamma \vdash_{l_w} e_2 : \langle \mathsf{E}, l \rangle \textit{ w.r.t. } \Sigma$$

  By inspection of the evaluation relation, either E-EQ1 or E-EQ2 may apply, where e_1 will be evaluated first, followed by e_2.

  By the induction hypothesis, either e_1 is a value, or there exists some state $\langle \Delta'', v_1 \rangle$ such that:

  $$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta'', v_1 \rangle$$

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

  $$\vdash_{l_w} v_1 : \langle \mathsf{E}, l \rangle \textit{ w.r.t. } \Sigma$$

  If e_1 is a value, we have:

  $$v_1 \;=\; e_1$$

  $$\Delta'' \;=\; \Delta \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

  By Lemma A.5 (Lower Bound on Security Level of Non-Ciphertext Types):

  $$l_w \leq l$$

  By a second application of the induction hypothesis, either e_2 is a value, or there exists some state $\langle \Delta''', v_2 \rangle$ such that:

  $$\Pi \vdash \langle \Delta'', e_2 \rangle \Downarrow \langle \Delta''', v_2 \rangle$$

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$

  $$\vdash_{l_w} v_2 : \langle \mathsf{E}, l \rangle \textit{ w.r.t. } \Sigma$$

  If e_2 is a value, we have:

  $$v_2 \;=\; e_2$$

  $$\Delta''' \;=\; \Delta'' \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$

  1. If $v_1 = v_2$ then E-EQ1 applies and we get:

     $$\underline{v = \text{TRUE}}$$

     $$\underline{\Delta' = \Delta'''} \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$$

By T-TRUE:

$$\vdash_l \text{TRUE} : \langle \mathsf{bool}, l \rangle \textit{ w.r.t. } \Sigma$$

By T-SUB and S-REFL:

$$\vdash_{l_w} \text{TRUE} : \langle \mathsf{bool}, l \rangle \textit{ w.r.t. } \Sigma \quad \equiv \quad \underline{\vdash_{l_w} v : \mathsf{S} \textit{ w.r.t. } \Sigma}$$

2. If $v_1 \neq v_2$ then E-EQ2 applies and we get:

$$\underline{v = \text{FALSE}}$$
$$\underline{\Delta' = \Delta'''} \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$$

By T-FALSE:

$$\vdash_l \text{FALSE} : \langle \mathsf{bool}, l \rangle \textit{ w.r.t. } \Sigma$$

By T-SUB and S-REFL:

$$\vdash_{l_w} \text{FALSE} : \langle \mathsf{bool}, l \rangle \textit{ w.r.t. } \Sigma \quad \equiv \quad \underline{\vdash_{l_w} v : \mathsf{S} \textit{ w.r.t. } \Sigma}$$

- Case T-ASGN:

We have:

$$e \;=\; e_1 := e_2$$
$$\Gamma \vdash_{l_w} e_1 : \langle \mathsf{S\,loc}, \top \rangle \textit{ w.r.t. } \Sigma$$
$$\Gamma \vdash_{l_w} e_2 : \mathsf{S} \textit{ w.r.t. } \Sigma$$

By inspection of the evaluation relation, only E-ASGN may apply, in which case e_1 will be evaluated first, followed by e_2. By the induction hypothesis, either e_1 is a value, or there exists some state $\langle \Delta'', v_1 \rangle$ such that:

$$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta'', v_1 \rangle$$
$$(\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$
$$\vdash_{l_w} v_1 : \langle \mathsf{S\,loc}, \top \rangle \textit{ w.r.t. } \Sigma$$

If e_1 is a value, we have:

$$v_1 \;=\; e_1$$
$$\Delta'' \;=\; \Delta \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

By a second application of the induction hypothesis, either e_2 is a value, or there exists some state $\langle \Delta''', v_2 \rangle$ such that:

$$\Pi \vdash \langle \Delta'', e_2 \rangle \Downarrow \langle \Delta''', v_2 \rangle$$
$$(\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$
$$\vdash_{l_w} v_2 : \mathsf{S} \textit{ w.r.t. } \Sigma$$

If e_2 is a value, we have:

$$v_2 \;=\; e_2$$

$$\Delta''' \;=\; \Delta'' \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta''')$$

By Lemma A.3 (Canonical Forms):

$$v_1 \;=\; a$$

Consequently, E-ASGN does apply and we get:

$$\underline{v = v_2} \quad \Rightarrow \quad \underline{\vdash_{l_w} v : \mathsf{S} \textit{ w.r.t. } \Sigma}$$

$$\underline{\Delta' = \Delta'''[a \mapsto v\,]}$$

By T-SUB and S-REFL:

$$\vdash_{\perp} v_2 : \mathsf{S} \textit{ w.r.t. } \Sigma \qquad \equiv \qquad \vdash_{\perp} v : \mathsf{S} \textit{ w.r.t. } \Sigma$$

By Lemma A.2 (Inversion of the Typing Relation):

$$a : \langle \mathsf{S}\,\mathsf{loc}, l \rangle \in \Sigma$$

By Definition 4.15 (Domain Consistency):

$$(\Gamma, \Sigma) \simeq (\Pi, \Delta'''[a \mapsto v\,]) \qquad \equiv \qquad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$$

- Case T-DREF:

  We have:

  $$e \;=\; *e'$$

  $$\mathsf{S} = \langle \mathsf{E}, l \curlyvee l_w \rangle$$

  $$\Gamma \vdash_{l_w} e' : \langle \langle \mathsf{E}, l \rangle\,\mathsf{loc}, \top \rangle \textit{ w.r.t. } \Sigma$$

  $$l_w \leq \lfloor \langle \mathsf{E}, l \curlyvee l_w \rangle \rfloor$$

  By inspection of the evaluation relation, only E-DREF may apply. By the induction hypothesis, either e' is a value, or there exists some state $\langle \Delta'', v' \rangle$ such that:

  $$\Pi \vdash \langle \Delta, e' \rangle \Downarrow \langle \Delta'', v' \rangle$$

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

  $$\vdash_{l_w} v' : \langle \langle \mathsf{E}, l \rangle\,\mathsf{loc}, \top \rangle \textit{ w.r.t. } \Sigma$$

  If e' is a value, we have:

  $$v' \;=\; e'$$

  $$\Delta'' \;=\; \Delta \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta'')$$

  By Lemma A.3 (Canonical Forms):

  $$v' \;=\; a$$

Consequently, E-DREF does apply and we get:

$$\underline{v = \Delta''(a)}$$
$$\underline{\Delta' = \Delta''} \quad \Rightarrow \quad \underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$$

By Lemma A.2 (Inversion of the Typing Relation):

$$a : \langle \langle \mathsf{E}, l \rangle \, \mathsf{loc}, l' \rangle \in \Sigma$$
$$l_w \leq \lfloor \langle \mathsf{E}, l \rangle \rfloor$$

By Definition 4.15 (Domain Consistency):

$$\vdash_{\perp} \Delta''(a) : \langle \mathsf{E}, l \rangle \textit{ w.r.t. } \Sigma$$

By Lemma A.4 (Sufficient Minimum Write Level):

$$\vdash_{\lfloor \langle \mathsf{E}, l \rangle \rfloor} v : \langle \mathsf{E}, l \rangle \textit{ w.r.t. } \Sigma$$

By T-SUB:

$$\vdash_{l_w} v : \langle \mathsf{E}, l \curlyvee l_w \rangle \textit{ w.r.t. } \Sigma \qquad \equiv \qquad \underline{\vdash_{l_w} v : \mathsf{S} \textit{ w.r.t. } \Sigma}$$

- Case T-FNAPP:

  We have:

  $$e \;=\; f(e_i{}^{\,i \in 1..n})$$
  $$f : \{ i : \mathsf{S}_i{}^{\,i \in 1..n} \} \xrightarrow{l'_w} \langle \mathsf{E}, l \rangle \in \Sigma$$
  $$l_w \leq l'_w$$
  $$\forall i \in 1..n \;\; \Gamma \vdash_{l_w} e_i : \mathsf{S}_i$$

  By inspection of the evaluation relation, only E-FNAPP may apply, in which case the parameter expressions (if present) are evaluated in left-to-right order, followed by the function body. If $n = 0$ then we proceed straight to dealing with the function body, otherwise we must first consider the parameter expressions.

  By the induction hypothesis, either e_1 is a value, or there exists some state $\langle \Delta_1, v_1 \rangle$ such that:

  $$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta_1, v_1 \rangle$$
  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1)$$
  $$\vdash_{l_w} v_1 : \mathsf{S}_1 \textit{ w.r.t. } \Sigma$$

  If e_1 is a value, we have:

  $$v_1 \;=\; e_1$$
  $$\Delta_1 \;=\; \Delta \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta_1)$$

The induction hypothesis can therefore be applied to each parameter in order, giving us:

$$\left.\begin{array}{l}\Pi \vdash \langle \Delta_{i-1}, e_i\rangle \Downarrow \langle \Delta_i, v_i\rangle \\ (\Gamma, \Sigma) \simeq (\Pi, \Delta_i) \\ \vdash_{l_w} v_i : \mathsf{S}_i \textit{ w.r.t. } \Sigma\end{array}\right\} \forall i{\in}1..n \text{ (where } \Delta_0 = \Delta)$$

By Definition 4.15 (Domain Consistency):

$$\left.\begin{array}{l} f(x_i{}^{i\in 1..n}) = e_f \in \Delta_n \\ (\Gamma, x_i' : \mathsf{S}_i) \vdash_{l_w'} [x_i'/x_i]e_f : \mathsf{S}\end{array}\right\} \text{where } x_i' \notin \mathsf{dom}(\Gamma) \text{ and } x_i' \notin FV(e_f)$$

By T-SUB and S-REFL:

$$\forall i{\in}1..n \ \vdash_{\perp} v_i : \mathsf{S}_i \textit{ w.r.t. } \Sigma$$

By Definition 4.15 (Domain Consistency):

$$x_i' \notin \mathsf{dom}(\Pi)$$

$$((\Gamma, x_i' : \mathsf{S}_i), \Sigma) \simeq ((\Pi \cup x_i' \mapsto v_i), \Delta_n)$$

By Assumption 4.16, either e_f is a value, or there exists some state $\langle \Delta_r, v\rangle$ such that:

$$(\Pi \cup x_i' \mapsto v_i) \vdash \langle \Delta, e_f\rangle \Downarrow \langle \Delta_r, v\rangle$$

$$((\Gamma, x_i' : \mathsf{S}_i), \Sigma) \simeq ((\Pi \cup x_i' \mapsto v_i), \Delta_r) \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta_r)$$

$$\vdash_{l_w'} v : \mathsf{S} \textit{ w.r.t. } \Sigma$$

If e_f is a value, we have:

$$v = e_f \quad \Rightarrow \quad \vdash_{l_w'} v : \mathsf{S} \textit{ w.r.t. } \Sigma$$

$$\Delta_r = \Delta_n \quad \Rightarrow \quad (\Gamma, \Sigma) \simeq (\Pi, \Delta_r)$$

In both cases, we have $\underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta_r)}$ and $\vdash_{l_w'} v : \mathsf{S}$ *w.r.t.* Σ. The final part of the result therefore follows from T-SUB and S-REFL:

$$\underline{\vdash_{l_w} v : \mathsf{S} \textit{ w.r.t. } \Sigma}$$

- Case T-SUB:

  We have:

  $$\Gamma' \vdash_{l_w'} e : \mathsf{S}' \textit{ w.r.t. } \Sigma$$

  $$\Gamma' \subseteq \Gamma$$

  $$\mathsf{S}' <: \mathsf{S}$$

  $$l_w \leq l_w'$$

If e is not a value, it follows from the induction hypothesis that there exists some state $\langle \Delta', v \rangle$ such that:

$$\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta', v \rangle$$

$$\underline{(\Gamma, \Sigma) \simeq (\Pi, \Delta')}$$

$$\vdash_{l'_w} v : \mathsf{S}' \textit{ w.r.t. } \Sigma$$

By T-SUB:

$$\underline{\vdash_{l_w} v : \mathsf{S} \textit{ w.r.t. } \Sigma}$$

□

A.2 Non-Interference

Here, we present the definitions and full proofs of the lemmas and theorems which are required for the proof of Theorem 4.20 (Non-Interferent Expression).

A.2.1 Definitions

Definition A.6 (Typed Indistinguishability of Values)**.**

Two values, v and v', are defined to be indistinguishable *at the observation level l with type S, written $v \approx_l v' : S$, when one or more of the following conditions hold:*

- $v = v'$
- $v \approx_l v' : \langle \mathsf{data}, l' \rangle \quad \textit{if} \quad l' > l$
- $v \approx_l v' : \langle \mathsf{bool}, l' \rangle \quad \textit{if} \quad l' > l$
- $v \approx_l v' : \langle S\ \mathsf{loc}, l' \rangle \quad \textit{if} \quad l' > l$ *and* $(\forall v_1, v_2 \ \textit{s.t.} \ \vdash_\perp v_1 : S$ *and* $\vdash_\perp v_2 : S,\ v_1 \approx_l v_2 : S)^a$
- $k \approx_l k' : \langle l_1\ l_2\ \mathsf{key}, l_3 \rangle \quad \textit{if} \quad l_1 > l$
- $\mathit{ctxt}(n, k, v_m) \approx_l \mathit{ctxt}(n', k', v'_m) : \langle \mathsf{enc}(\langle E_m, l_m \rangle), l' \rangle \quad \textit{if} \quad k \approx_l k' : \langle \lfloor \langle E_m, l_m \rangle \rfloor\ l_m\ \mathsf{key}, \top \rangle$ *and* $v_m \approx_l v'_m : \langle E_m, l_m \rangle$

[a] or equivalently: $v \approx_l v' : \langle \mathsf{S\ loc}, l' \rangle$ if $l' > l$ and $\lfloor \mathsf{S} \rfloor > l$ (since we get $\vdash_{\lfloor \mathsf{S} \rfloor} v_1 : \mathsf{S}$ and $\vdash_{\lfloor \mathsf{S} \rfloor} v_2 : \mathsf{S}$ via Lemma A.4, and $v_1 \approx_l v_2 : \mathsf{S}$ via Lemma 4.24)

Definition A.7 (Typed Indistinguishability of Evaluation Environments)**.**

With respect to some typing environment Σ, two evaluation environments Δ_1 and Δ_2, are defined to be indistinguishable *at the observation level l, written $\Sigma \vdash \Delta_1 \approx_l \Delta_2$, when both of the following rules hold:*

- $\mathit{dom}(\Delta_1) = \mathit{dom}(\Delta_2)$
- $((a \mapsto v_1) \in \Delta_1 \wedge (a \mapsto v_2) \in \Delta_2) \rightarrow v_1 \approx_l v_2 : S \qquad (\textit{where } a : \langle S\ \mathsf{loc}, l' \rangle \in \Sigma)$
- $f(x_i{}^{\,i \in 1..n}) = e_f \in \Delta_1 \quad \leftrightarrow \quad f(x_i{}^{\,i \in 1..n}) = e_f \in \Delta_2$

Definition A.8 (Typed Indistinguishability of Evaluation Contexts)**.**

With respect to some typing context Γ, two evaluation contexts Π_1 and Π_2, are defined to be indistinguishable *at the observation level l, written $\Gamma \vdash \Pi_1 \approx_l \Pi_2$, when both of the following rules hold:*

- $((x \mapsto v_1) \in \Pi_1 \wedge (x \mapsto v_2) \in \Pi_2) \rightarrow v_1 \approx_l v_2 : S \qquad (\textit{where } x : S \in \Gamma)$

A.2.2 Lemmas

Lemma A.9 (Widening of Typed Indistinguishability for Values)**.**
If $v \approx_l v' : \mathsf{S}$, $l' \leq l$ *and* $\mathsf{S} <: \mathsf{S}'$ *then* $v \approx_{l'} v' : \mathsf{S}'$.

Proof. By case-analysis on the rules which define $v \approx_l v' : \mathsf{S}$.

- Case $v = v'$

  This rule does not take account of l or S, therefore this case still applies and the result holds.

- Case $v \approx_l v' : \langle \mathsf{data}, l'' \rangle$ where $l'' > l$

  By inspection of the subtyping rules, we get that $\mathsf{S}' = \langle \mathsf{data}, l''' \rangle$ where $l''' \geq l''$. Consequently, it follows from the transitivity of $>$ and $\geq$ that $l''' > l'$, therefore this case still applies and the result holds.

- Case $v \approx_l v' : \langle \mathsf{bool}, l'' \rangle$ where $l'' > l$

  By inspection of the subtyping rules, we get that $\mathsf{S}' = \langle \mathsf{bool}, l''' \rangle$ where $l''' \geq l''$. Consequently, it follows from the transitivity of $>$ and $\geq$ that $l''' > l'$, therefore this case still applies and the result holds.

- Case $v \approx_l v' : \langle \mathsf{S}'' \, \mathsf{loc}, l_k \rangle$ where $l'' > l$ and $v_1 \approx_l v_2 : \mathsf{S}''$

  By inspection of the subtyping rules, we get that $\mathsf{S}' = \langle \mathsf{S}'' \, \mathsf{loc}, l''' \rangle$ where $l''' \geq l''$. Consequently, it follows from the transitivity of $>$ and $\geq$ that $l''' > l'$, therefore this case still applies and the result holds.

- Case $k \approx_l k' : \langle l_1 \, l_2 \, \mathsf{key}, l_3 \rangle$ where $l_1 > l$

  By inspection of the subtyping rules, we get that $\mathsf{S}' = \langle l_1 \, l_2 \, \mathsf{key}, l_3' \rangle$, therefore this case still applies and the result holds.

- Case $\mathrm{ctxt}(n, k, v_m) \approx_l \mathrm{ctxt}(n', k', v_m') : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), l'' \rangle$ where $k \approx_l k' : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \, l_m \, \mathsf{key}, \top \rangle$ and $v_m \approx_l v_m' : \langle \mathsf{E}_m, l_m \rangle$:

  By inspection of the subtyping rules, $\mathsf{S}' = \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), l''' \rangle$, where $l'' \leq l'''$. By the induction hypothesis, we get $k \approx_{l'} k' : \mathsf{S}_k$ where $\langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \, l_m \, \mathsf{key}, \top \rangle <: \mathsf{S}_k$. By inspection of the subtyping rules, $\mathsf{S}_k = \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \, l_m \, \mathsf{key}, \top \rangle$. By inspection of Definition A.6 (Typed Indistinguishability of Values), we must have either $k = k'$ or $l' < \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor$. In the latter case we get $\lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \leq l_m$

via Definition 4.13 therefore $l' < l_m$ follows via the transitivity of $<$ and $\leq$, thus $k \approx_{l'} k' : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \, l_m \, \mathsf{key}, \top \rangle$ holds in both cases. By another application of the induction hypothesis, we get $v_m \approx_{l'} v'_m : \langle \mathsf{E}_m, l_m \rangle$. We therefore get $\mathrm{ctxt}(n, k, v_m) \approx_{l'} \mathrm{ctxt}(n', k', v'_m) : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), l''' \rangle$ from the rule in this case and thus the result holds. □

Lemma 4.24 (Typed Indistinguishability of Arbitrary Values)

If $\vdash_{l_w} v : \mathsf{S}$ *w.r.t.* Σ, $\vdash_{l_w} v' : \mathsf{S}$ *w.r.t.* Σ, Σ *is valid and* $l < l_w$ *then* $v \approx_l v' : \mathsf{S}$.

Proof. By induction on the form of S.

- Cases $\langle \mathsf{data}, l' \rangle$ and $\langle \mathsf{bool}, l' \rangle$:

  By Lemma A.5 (Lower Bound on the Security Level of Non-Ciphertext Types):

  $$l_w \leq l' \quad \Rightarrow \quad l < l'$$

  Consequently, the result follows from the second or third rule in Definition A.6 (Typed Indistinguishability of Values).

- Case $\langle \mathsf{S}' \, \mathsf{loc}, l' \rangle$:

  By Lemma A.5 (Lower Bound on the Security Level of Non-Ciphertext Types):

  $$l_w \leq l' \quad \Rightarrow \quad l < l'$$

  By Lemma A.3 (Canonical Forms):

  $$v = a$$

  By Lemma A.2 (Inversion of the Typing Relation):

  $$a : \langle \mathsf{S}' \, \mathsf{loc}, l'' \rangle \in \Sigma$$
  $$l_w \leq \lfloor \mathsf{S}' \rfloor \qquad \Rightarrow \qquad l < \lfloor \mathsf{S}' \rfloor$$

  Taking any v_1 and v_2 such that $\vdash_\perp v_1 : \mathsf{S}'$ and $\vdash_\perp v_2 : \mathsf{S}'$, it follows from Lemma A.4 (Sufficient Minimum Write Level) that:

  $$\vdash_{\lfloor \mathsf{S}' \rfloor} v_1 : \mathsf{S}'$$
  $$\vdash_{\lfloor \mathsf{S}' \rfloor} v_2 : \mathsf{S}'$$

  By the induction hypothesis:

  $$v_1 \approx_l v_2 : \mathsf{S}'$$

  Consequently, the result follows from the fourth rule in Definition A.6 (Typed Indistinguishability of Values).

- Case $\langle l_1\ l_2\ \mathsf{key}, l_3\rangle$:

  By Lemma A.3 (Canonical Forms):

  $$v = k$$
  $$v' = k'$$

  By Lemma A.2 (Inversion of the Typing Relation):

  $$l_w \leq l_1 \quad \Rightarrow \quad l < l_1$$

  Consequently, the result follows from the fifth rule in Definition A.6 (Typed Indistinguishability of Values).

- Case $\langle \mathsf{enc}(\langle \mathsf{E}_m, l_m\rangle), l'\rangle$:

  By Lemma A.3 (Canonical Forms):

  $$v = \mathrm{ctxt}(n, k, v_m)$$
  $$v' = \mathrm{ctxt}(n', k', v'_m)$$

  By Lemma A.2 (Inversion of the Typing Relation):

  $$\vdash_{l_w} k : \langle \lfloor \langle \mathsf{E}_m, l_m\rangle \rfloor\ l_m\ \mathsf{key}, \top\rangle \qquad \vdash_{l_w} k' : \langle \lfloor \langle \mathsf{E}_m, l_m\rangle \rfloor\ l_m\ \mathsf{key}, \top\rangle$$
  $$\vdash_{l_w} v_m : \langle \mathsf{E}_m, l_m\rangle \qquad \text{and} \qquad \vdash_{l_w} v'_m : \langle \mathsf{E}_m, l_m\rangle$$

  By the induction hypothesis:

  $$k \approx_l k' : \langle \lfloor \langle \mathsf{E}_m, l_m\rangle \rfloor\ l_m\ \mathsf{key}, \top\rangle$$
  $$v_m \approx_l v'_m : \langle \mathsf{E}_m, l_m\rangle$$

  Consequently, the result follows from the final rule in Definition A.6 (Typed Indistinguishability of Values). □

Lemma A.10 (Transitivity of Typed Indistinguishability for Values)**.**

If $v_1 \approx_l v_2 : \mathsf{S}$ *and* $v_2 \approx_l v_3 : \mathsf{S}$ *then* $v_1 \approx_l v_3 : \mathsf{S}$.

Proof.

By induction over the rules which define $v_1 \approx_l v_2 : \mathsf{S}$:

- Case $v_1 = v_2$:

  Here, $v_2 \approx_l v_3 : \mathsf{S}$ is equivalent to $v_1 \approx_l v_3 : \mathsf{S}$, therefore the result is immediate.

- Cases $\mathsf{S} \in \{\langle \mathsf{data}, l' \rangle, \langle \mathsf{bool}, l' \rangle, \langle \mathsf{S}' \ \mathsf{loc}, l' \rangle, \langle l_1 \ l_2 \ \mathsf{key}, l_3 \rangle\}$:

  In these cases, the relation only depends upon the observation level and type, not the specific values, therefore the result follows from an application of the same rule.

- Case $\mathsf{S} = \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), l' \rangle$:

  Here, we have:

  $$v_1 = \mathrm{ctxt}(n_1, k_1, v_1')$$
  $$v_2 = \mathrm{ctxt}(n_2, k_2, v_2')$$
  $$v_3 = \mathrm{ctxt}(n_3, k_3, v_3')$$
  $$k_1 \approx_l k_2 : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \ l_m \ \mathsf{key}, \top \rangle$$
  $$k_2 \approx_l k_3 : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \ l_m \ \mathsf{key}, \top \rangle$$
  $$v_1' \approx_l v_2' : \langle \mathsf{E}_m, l_m \rangle$$
  $$v_2' \approx_l v_3' : \langle \mathsf{E}_m, l_m \rangle$$

  By the induction hypothesis, we get:

  $$k_1 \approx_l k_3 : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \ l_m \ \mathsf{key}, \top \rangle$$
  $$v_1' \approx_l v_3' : \langle \mathsf{E}_m, l_m \rangle$$

  Consequently, the result follows from the final rule in Definition A.6 (Typed Indistinguishability for Values). □

Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments)**.**

If $\Sigma \vdash \Delta_1 \approx_l \Delta_2$ *and* $\Sigma \vdash \Delta_2 \approx_l \Delta_3$ *then* $\Sigma \vdash \Delta_1 \approx_l \Delta_3$.

Proof. By inspection of Definition A.7 (Typed Indistinguishability of Evaluation Environments):

$$\left.\begin{array}{l} \mathsf{dom}(\Pi_1) = \mathsf{dom}(\Pi_2) \\ \mathsf{dom}(\Pi_2) = \mathsf{dom}(\Pi_3) \end{array}\right\} \Rightarrow \quad \underline{\mathsf{dom}(\Pi_1) = \mathsf{dom}(\Pi_3)}$$

$$((a \mapsto v_1) \in \Delta_1 \wedge (a \mapsto v_2) \in \Delta_2) \rightarrow v_1 \approx_l v_2 : \mathsf{S} \qquad (\text{where } a : \langle \mathsf{S} \ \mathsf{loc}, l' \rangle \in \Sigma)$$

$$((a' \mapsto v_2') \in \Delta_2 \wedge (a' \mapsto v_3') \in \Delta_3) \rightarrow v_2' \approx_l v_3' : \mathsf{S}' \qquad (\text{where } a' : \langle \mathsf{S}' \ \mathsf{loc}, l'' \rangle \in \Sigma)$$

$$f(x_i \ ^{i \in 1..n}) = e_f \in \Delta_1 \quad \leftrightarrow \quad f(x_i \ ^{i \in 1..n}) = e_f \in \Delta_2$$

$$f(x_i \ ^{i \in 1..n}) = e_f \in \Delta_2 \quad \leftrightarrow \quad f(x_i \ ^{i \in 1..n}) = e_f \in \Delta_3$$

It therefore follows from the transitivity of $\leftrightarrow$ that:

$$\underline{f(x_i \ ^{i \in 1..n}) = e_f \in \Delta_1 \quad \leftrightarrow \quad f(x_i \ ^{i \in 1..n}) = e_f \in \Delta_3}$$

It therefore follows from the equality of domains that, $\forall a \in \mathsf{dom}(\Delta_i)$:

$$\left.\begin{array}{l}\Delta_1[a] \approx_l \Delta_2[a] : \mathsf{S} \\ \Delta_2[a] \approx_l \Delta_3[a] : \mathsf{S}\end{array}\right\} \quad (\text{where } a : \langle \mathsf{S}\,\mathsf{loc}, l' \rangle \in \Sigma)$$

Finally, by Lemma A.10 (Transitivity of Typed Indistinguishability for Values):

$$\underline{\forall a \in \mathsf{dom}(\Delta_1)\ \ \Delta_1[a] \approx_l \Delta_3[a] : \mathsf{S}} \qquad (\text{where } a : \langle \mathsf{S}\,\mathsf{loc}, l' \rangle \in \Sigma)$$

Lemma A.12 (Preservation Under Evaluation of Typed Indistinguishability for Evaluation Environments)**.**

If $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$, $\Gamma \vdash_{l_w} e : S$ *w.r.t.* Σ *and* $\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta_r, v \rangle$ *then* $\Sigma \vdash \Delta \approx_l \Delta_r$ *for all* $l < l_w$.

Proof. By induction on the structure of the derivation of $\Gamma \vdash_{l_w} e : \mathsf{S}$ *w.r.t.* Σ.

- Cases T-Cnst, T-True, T-False, T-Loc, T-Key and T-Ctxt:

  Here, e is a value therefore, by inspection of the evaluation rules, E-Val must be at the root of the evaluation, thus $\Delta_r = \Delta$ and the result holds.

- Case T-Var:

  Here, e is a variable therefore, by inspection of the evaluation rules, E-Var must be at the root of the evaluation, thus $\Delta_r = \Delta$ and the result holds.

- Case T-Let:

  Here, we have:

  $$e = \text{let } x = e_1 \text{ in } e_2$$
  $$\Gamma \vdash_{l_w} e_1 : \mathsf{S}'$$
  $$x \notin \mathsf{dom}(\Sigma)$$
  $$(\Gamma, x : \mathsf{S}') \vdash_{l_w} e_2 : \mathsf{S}$$

  By inspection of the evaluation rules, we must have E-Let at the root of the evaluation, thus:

  $$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta_1, v_1 \rangle$$
  $$\Pi \cup (x \mapsto v_1) \vdash \langle \Delta_1, e_2 \rangle \Downarrow \langle \Delta_r, v \rangle$$

  By the induction hypothesis:

  $$\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_1$$

  By Theorem 4.18 (Type Safety):

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1)$$

By Definition 4.15 (Domain Consistency):

$$((\Gamma, x : \mathsf{S}'), \Sigma) \simeq (\Pi \cup (x \mapsto v_1), \Delta_1)$$

By a second application of the induction hypothesis:

$$\forall l < l_w.\ \Sigma \vdash \Delta_1 \approx_l \Delta_r$$

By Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

$$\underline{\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_r}$$

- Case T-SENC:

  Here, we have:

  $$e \ = \ \mathrm{senc}(e_k, e_m)$$

  $$\Gamma \vdash_{l_w} e_k : \langle l'\ l_m\ \mathsf{key}, \top \rangle$$

  $$\Gamma \vdash_{l_w} e_m : \langle \mathsf{E}_m, l_m \rangle$$

  By inspection of the evaluation rules, we must have E-SENC at the root of the evaluation, thus:

  $$\Pi \vdash \langle \Delta, e_k \rangle \Downarrow \langle \Delta_1, k \rangle$$

  $$\Pi \vdash \langle \Delta_1, e_m \rangle \Downarrow \langle \Delta_2, v_m \rangle$$

  $$\Delta_r = \Delta_2 \cup n$$

  $$n \notin \Delta_2$$

  By the induction hypothesis:

  $$\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_1$$

  By Theorem 4.18 (Type Safety):

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1)$$

  By a second application of the induction hypothesis:

  $$\forall l < l_w.\ \Sigma \vdash \Delta_1 \approx_l \Delta_2$$

  By Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

  $$\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_2$$

  By Definition A.7 (Typed Indistinguishability of Evaluation Environments):

  $$\forall l < l_w.\ \Sigma \vdash \Delta \approx_l (\Delta_2 \cup n) \qquad \equiv \qquad \underline{\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_r}$$

- Case T-SDEC:

  Here, we have:

  $$e \;=\; \text{try } \mathrm{sdec}(e_k, e_c) = x \text{ in } e_1 \text{ else } e_2$$

  $$\Gamma \vdash_{l_w} e_k : \langle l'\; l_m\; \mathsf{key}, \top \rangle$$

  $$\Gamma \vdash_{l_w} e_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle$$

  $$l' \leq \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor$$

  $$x \notin \mathsf{dom}(\Sigma)$$

  $$(\Gamma, x : \langle \mathsf{E}_m, l_m \rangle) \vdash_{l' \curlyvee l_w} e_1 : \mathsf{S}$$

  $$\Gamma \vdash_{l' \curlyvee l_w} e_2 : \mathsf{S}$$

  By inspection of the evaluation rules, either E-SDEC1 or E-SDEC2 may be at the root of the evaluation. In both cases, the key and ciphertext expressions are evaluated in that order, therefore we must have:

  $$\Pi \vdash \langle \Delta, e_k \rangle \Downarrow \langle \Delta_1, k \rangle$$

  $$\Pi \vdash \langle \Delta_1, e_c \rangle \Downarrow \langle \Delta_2, v_c \rangle$$

  By the induction hypothesis:

  $$\forall l < l_w.\; \Sigma \vdash \Delta \approx_l \Delta_1$$

  By Theorem 4.18 (Type Safety):

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1)$$

  By a second application of the induction hypothesis:

  $$\forall l < l_w.\; \Sigma \vdash \Delta_1 \approx_l \Delta_2$$

  By Theorem 4.18 (Type Safety):

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_2)$$

  By the properties of $\curlyvee$:

  $$l_w \leq (l' \curlyvee l_w) \qquad \Rightarrow \qquad l < (l' \curlyvee l_w)$$

  We now split on which rule is at the root of the evaluation:

  - Subcase E-SDEC1:

    In this subcase, we have:

    $$v_c \;=\; \mathrm{ctxt}(n, k, v_m)$$

    $$\Pi \cup (x \mapsto v_m) \vdash \langle \Delta_2, e_1 \rangle \Downarrow \langle \Delta_r, v \rangle$$

By Theorem 4.18 (Type Safety):

$$\vdash_{l_w} v_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle$$

By Lemma A.1 (Inversion of the Typing Relation):

$$\vdash_{l_w} v_m : \langle \mathsf{E}_m, l_m \rangle$$

By Definition 4.15 (Domain Consistency):

$$((\Gamma, x : \langle \mathsf{E}_m, l_m \rangle), \Sigma) \simeq (\Pi \cup (x \mapsto v_m), \Delta_2)$$

By the induction hypothesis:

$$\forall l < (l' \curlyvee l_w).\ \Sigma \vdash \Delta_2 \approx_l \Delta_r \qquad \Rightarrow \qquad \forall l < l_w.\ \Sigma \vdash \Delta_2 \approx_l \Delta_r$$

By Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

$$\underline{\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_r}$$

- Subcase E-SDEC2:

  In this subcase, we have:

  $$\Pi \vdash \langle \Delta_2, e_2 \rangle \Downarrow \langle \Delta_r, v \rangle$$

  By the induction hypothesis:

  $$\forall l < (l' \curlyvee l_w).\ \Sigma \vdash \Delta_2 \approx_l \Delta_r \qquad \Rightarrow \qquad \forall l < l_w.\ \Sigma \vdash \Delta_2 \approx_l \Delta_r$$

  By Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

  $$\underline{\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_r}$$

- Case T-IF:

  Here, we have:

  $$e \ = \ \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

  $$\Gamma \vdash_{l_w} e_1 : \langle \mathsf{bool}, l' \rangle$$

  $$\Gamma \vdash_{l' \curlyvee l_w} e_2 : \mathsf{S}$$

  $$\Gamma \vdash_{l' \curlyvee l_w} e_3 : \mathsf{S}$$

  By inspection of the evaluation rules, either E-IF1 or E-IF2 may be at the root of the evaluation. In both cases, the guard expression will have been evaluated first, therefore we must have:

  $$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta_1, v_1 \rangle$$

By the induction hypothesis:

$$\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_1$$

By Theorem 4.18 (Type Safety):

$$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1)$$

By the properties of $\curlyvee$:

$$l_w \leq (l' \curlyvee l_w) \qquad \Rightarrow \qquad l < (l' \curlyvee l_w)$$

If the rule at the root of the evaluation is E-IF1, we have:

$$\Pi \vdash \langle \Delta_1, e_2 \rangle \Downarrow \langle \Delta_r, v \rangle$$

Otherwise, the rule at the root of the evaluation is E-IF2 and we have:

$$\Pi \vdash \langle \Delta_1, e_3 \rangle \Downarrow \langle \Delta_r, v \rangle$$

In both cases, it follows from the induction hypothesis that:

$$\forall l < (l' \curlyvee l_w).\ \Sigma \vdash \Delta_1 \approx_l \Delta_r \qquad \Rightarrow \qquad \forall l < l_w.\ \Sigma \vdash \Delta_1 \approx_l \Delta_r$$

By Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

$$\underline{\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_r}$$

- Case T-EQ:

  Here, we have:

  $$e \ = \ e_1 == e_2$$

  $$\Gamma \vdash_{l_w} e_1 : \langle \mathsf{E}, l' \rangle$$

  $$\Gamma \vdash_{l_w} e_2 : \langle \mathsf{E}, l' \rangle$$

  By inspection of the evaluation rules, either E-EQ1 or E-EQ2 may be at the root of the evaluation. In both cases, the two sub-expressions will have been evaluated in left-to-right order, therefore we must have:

  $$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta_1, v_1 \rangle$$

  $$\Pi \vdash \langle \Delta_1, e_2 \rangle \Downarrow \langle \Delta_r, v_2 \rangle$$

  By the induction hypothesis:

  $$\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_1$$

  By Theorem 4.18 (Type Safety):

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1)$$

By a second application of the induction hypothesis:

$$\forall l < l_w.\, \Sigma \vdash \Delta_1 \approx_l \Delta_2$$

By Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

$$\underline{\forall l < l_w.\, \Sigma \vdash \Delta \approx_l \Delta_r}$$

- Case T-ASGN:

  Here, we have:

  $$e \;=\; e_1 := e_2$$
  $$\Gamma \vdash_{l_w} e_1 : \langle \mathsf{S\ loc}, \top \rangle$$
  $$\Gamma \vdash_{l_w} e_2 : \mathsf{S}$$

  By inspection of the evaluation rules, we must have E-ASGN at the root of the evaluation, thus:

  $$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta_1, a \rangle$$
  $$\Pi \vdash \langle \Delta_1, e_2 \rangle \Downarrow \langle \Delta_2, v \rangle$$
  $$\Delta_r \;=\; \Delta_2[a \mapsto v]$$

  By the induction hypothesis:

  $$\forall l < l_w.\, \Sigma \vdash \Delta \approx_l \Delta_1$$

  By two applications of Theorem 4.18 (Type Safety):

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1) \qquad \vdash_{l_w} a : \langle \mathsf{S\ loc}, \top \rangle \qquad \text{and} \qquad \vdash_{l_w} v : \mathsf{S}$$

  By a second application of the induction hypothesis:

  $$\forall l < l_w.\, \Sigma \vdash \Delta_1 \approx_l \Delta_2$$

  By Lemma A.1 (Inversion of the Typing Relation):

  $$a : \langle \mathsf{S\ loc}, l' \rangle \in \Sigma$$
  $$l_w \leq \lfloor \mathsf{S} \rfloor \qquad \Rightarrow \qquad \forall l < l_w.\, l < \lfloor \mathsf{S} \rfloor$$

  By Definition 4.15 (Domain Consistency):

  $$\vdash_{\perp} \Delta[a] : \mathsf{S}$$

  By Lemma A.1 (Sufficient Minimum Write Level):

  $$\vdash_{\lfloor \mathsf{S} \rfloor} \Delta[a] : \mathsf{S}$$

By Lemma 4.24 (Typed Indistinguishability of Arbitrary Values):

$$\forall l < l_w.\ \Delta(a) \approx_l v : \mathsf{S}$$

By Definition A.7 (Typed Indistinguishability of Evaluation Environments):

$$\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_2[a \mapsto v] \qquad \equiv \qquad \underline{\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_r}$$

- Case T-DREF:

  Here, we have:

  $$e \ = \ *e'$$

  $$\Gamma \vdash_{l_w} e' : \mathsf{S}'$$

  By inspection of the evaluation rules, we must have E-DREF at the root of the evaluation, thus:

  $$\Pi \vdash \langle \Delta, e' \rangle \Downarrow \langle \Delta_r, a \rangle$$

  The result therefore follows from the induction hypothesis.

- Case T-FNAPP:

  Here, we have:

  $$e \ = \ f(e_i \ ^{i \in 1..n})$$

  $$f : \{ i : \mathsf{S}_i \ ^{i \in 1..n} \} \xrightarrow{l'_w} \langle \mathsf{E}, l \rangle \in \Sigma$$

  $$l_w \leq l'_w$$

  $$\forall i \in 1..n \ \ \Gamma \vdash_{l_w} e_i : \mathsf{S}_i$$

  By inspection of the evaluation rules, we must have E-FNAPP at the root of the evaluation, thus:

  $$\forall i \in 1..n \ \ \Pi \vdash \langle \Delta_{i-1}, e_i \rangle \Downarrow \langle \Delta_i, v_i \rangle$$

  $$f(x_i \ ^{i \in 1..n}) = \ e_f \in \Delta_n$$

  $$(\Pi \cup x'_i \mapsto v_i) \vdash \langle \Delta_n, [x'_i / x_i] e_f \rangle \Downarrow \langle \Delta_r, v \rangle$$

  where

  $$\Delta_0 = \Delta$$

  $$x'_i \notin \mathsf{dom}(\Pi)$$

  $$x'_i \notin FV(e_f)$$

  By n applications of Theorem 4.18 (Type Safety):

  $$\left.\begin{array}{ll} \vdash_{l_w} v_i : \mathsf{S}_i & \\ (\Gamma, \Sigma) \simeq (\Pi, \Delta_i) & \end{array}\right\} \ \forall i \in 1..n$$

By n applications of the induction hypothesis:

$$\forall l \leq l_w.\ \Sigma \vdash \Delta_{i-1} \approx_{l_o} \Delta_i \qquad \forall i \in 1..n$$

By T-SUB and S-REFL:

$$\left.\begin{array}{l} \vdash_{\perp} v_i : \mathsf{S}_i \text{ w.r.t. } \Sigma \\ \vdash_{\perp} v_i' : \mathsf{S}_i \text{ w.r.t. } \Sigma \end{array}\right\} \ \forall i \in 1..n$$

By Definition 4.15 (Domain Consistency):

$$x_i' \notin \mathsf{dom}(\Gamma)$$

$$(\Gamma, x_i' : \mathsf{S}_i) \vdash_{l_w'} [x_i'/x_i]e_f : \mathsf{S}$$

$$((\Gamma, x_i' : \mathsf{S}_i), \Sigma) \simeq ((\Pi \cup x_i' \mapsto v_i), \Delta_n)$$

By the induction hypothesis:

$$\forall l \leq l_w'.\ \Sigma \vdash \Delta_n \approx_{l_o} \Delta_r \qquad \Rightarrow \qquad \forall l \leq l_w.\ \Sigma \vdash \Delta_n \approx_{l_o} \Delta_r$$

By Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

$$\underline{\forall l < l_w.\ \Sigma \vdash \Delta \approx_l \Delta_r}$$

- Case T-SUB:

  Here, we have $\Gamma \vdash_{l_w'} e : \mathsf{S}'$ *w.r.t.* Σ and $l_w \leq l_w'$, therefore $l < l_w'$ and the result follows from the induction hypothesis. □

Lemma 4.22 (Indistinguishability)

If $\vdash_{l_w} v_1 : \langle E, l \rangle$ *w.r.t.* Σ, $\vdash_{l_w} v_2 : \langle E, l \rangle$ *w.r.t.* Σ *and* $v_1 \approx_{l_o} v_2 : \langle E, l \rangle$ *then, for any typing context* Γ *and evaluation domain* (Π, Δ) *where* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$*, it follows from* $l_o \geq l$ *that* $\Delta \vdash v_1 \approx_{l_o} v_2$.

Proof. By induction on the structure of the derivation of $\vdash_{l_w} v_1 : \langle \mathsf{E}, l \rangle$ w.r.t. Σ

- Case c:

  By Lemma A.2 (Inversion of the Typing Relation):

  $$\mathsf{E} = \mathsf{data}$$

  By Lemma A.3 (Canonical Forms):

  $$v_2 = c'$$

  By Definition A.6 (Typed Indistinguishability):

  $$v_1 = v_2 \quad \textit{or} \quad l_o < l$$

Since we have $l_o \geq l$ as a precondition, we must have $v_1 = v_2$. Consequently, the result follows from the first rule in Definition 4.8 (Indistinguishability of Values).

- Cases TRUE and FALSE:

  By Lemma A.2 (Inversion of the Typing Relation):

  $\mathsf{E} = \mathsf{bool}$

  By Lemma A.3 (Canonical Forms):

  $v_2 \in \{\text{TRUE}, \text{FALSE}\}$

  By Definition A.6 (Typed Indistinguishability):

  $v_1 = v_2 \quad or \quad l_o < l$

  Since we have $l_o \geq l$ as a precondition, we must have $v_1 = v_2$. Consequently, the result follows from the first rule in Definition 4.8 (Indistinguishability of Values).

- Case a:

  By Lemma A.2 (Inversion of the Typing Relation):

  $\mathsf{E} = \mathsf{S}\ \mathsf{loc}$

  By Lemma A.3 (Canonical Forms):

  $v_2 = a'$

  By Definition A.6 (Typed Indistinguishability):

  $v_1 = v_2$

  or

  $l_o < l$

  $\forall v, v'$ s.t. $\vdash_\perp v : \mathsf{S}$ and $\vdash_\perp v' : \mathsf{S}$, $v \approx_{l_o} v' : \mathsf{S}$

  Since we have $l_o \geq l$ as a precondition, we must have $v_1 = v_2$. Consequently, the result follows from the first rule in Definition 4.8 (Indistinguishability of Values).

- Case k:

  By Lemma A.2 (Inversion of the Typing Relation):

  $\mathsf{E} = l_1\ l_2\ \mathsf{key}$

  $l_1 \leq l_2 \leq l$

By Lemma A.3 (Canonical Forms):

$$v_2 = k'$$

By Definition A.6 (Typed Indistinguishability):

$$v_1 = v_2 \quad or \quad l_o < l_1$$

Since we have $l_o \geq l$ as a precondition, and $l \geq l_1$, we must have $v_1 = v_2$. Consequently, the result follows from the first rule in Definition 4.8 (Indistinguishability of Values).

- Case $\mathrm{ctxt}(n, k, v_m)$:

  By Lemma A.2 (Inversion of the Typing Relation):

  $$\mathsf{E} = \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle)$$

  By Lemma A.3 (Canonical Forms):

  $$v_2 = \mathrm{ctxt}(n', k', v'_m)$$

  By Definition A.6 (Typed Indistinguishability):

  $$v_1 = v_2$$

  $$or$$

  $$k \approx_{l_o} k' : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \, l_m \, \mathsf{key}, \top \rangle$$

  $$v_m \approx_{l_o} v'_m : \langle \mathsf{E}_m, l_m \rangle$$

  If $v_1 = v_2$ then the result follows immediately from the first rule in Definition 4.8 (Indistinguishability of Values).

  Otherwise, by two applications of Lemma A.2 (Inversion of the Typing Relation):

  $$\vdash_{l_w} k : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \, l_m \, \mathsf{key}, \top \rangle \; w.r.t. \; \Sigma$$

  $$\vdash_{l_w} k' : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \, l_m \, \mathsf{key}, \top \rangle \; w.r.t. \; \Sigma$$

  By two further applications of Lemma A.2 (Inversion of the Typing Relation):

  $$k : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \, l_m \, \mathsf{key}, l_k \rangle \in \Sigma$$

  $$l_m \leq l_k$$

  $$k' : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \, l_m \, \mathsf{key}, l'_k \rangle \in \Sigma$$

  $$l_m \leq l'_k$$

By Definition 4.15 (Domain Consistency):

$k : l_k \in \Delta$

$k' : l'_k \in \Delta$

By Definition 4.5 (Security Level Function):

$lvl_\Delta(k) = l_k \quad \Rightarrow \quad lvl_\Delta(k) \geq l_m$

$lvl_\Delta(k') = l'_k \quad \Rightarrow \quad lvl_\Delta(k') \geq l_m$

We now split on the relative ordering of l_o and l_m:

Subcase $l_o \geq l_m$:

By Definition A.6 (Typed Indistinguishability):

$k = k' \quad or \quad \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor > l_o$

By Lemma A.2 (Inversion of the Typing Relation):

$k : \langle \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \; l_m \; \mathsf{key}, l_k \rangle \in \Sigma$

$\lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \leq l_m \quad \Rightarrow \quad \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor \leq l_o$

Consequently, we must have $k = k'$, therefore the result follows from the third rule in Definition 4.8 (Indistinguishability of Values).

Subcase $l_o \not\geq l_m$:

Here, we have $lvl_\Delta(k) \not\leq l_o$ and $lvl_\Delta(k') \not\leq l_o$, therefore the result follows from the second rule in Definition 4.8 (Indistinguishability of Values). □

Lemma 4.23 (Indistinguishability of Evaluation Environments)

If $\Sigma \vdash \Delta_1 \approx_l \Delta_2$, $(\Gamma, \Sigma) \simeq (\Pi_1, \Delta_1)$ *and* $(\Gamma, \Sigma) \simeq (\Pi_2, \Delta_2)$ *then* $\Delta_1 \approx_l \Delta_2$.

Proof.

By Definition A.7 (Typed Indistinguishability of Evaluation Environments):

$\mathsf{dom}(\Delta_1) = \mathsf{dom}(\Delta_2)$

$((a \mapsto v_1) \in \Delta_1 \wedge (a \mapsto v_2) \in \Delta_2) \rightarrow v_1 \approx_l v_2 : \mathsf{S} \qquad (\text{where } a : \langle \mathsf{S}\,\mathsf{loc}, l' \rangle \in \Sigma)$

$f(x_i{}^{i \in 1..n}) = e_f \in \Delta_1 \;\leftrightarrow\; f(x_i{}^{i \in 1..n}) = e_f \in \Delta_2 \quad \Rightarrow \quad \mathbb{F}_{\Delta_1} = \mathbb{F}_{\Delta_2} = \mathbb{F}$

By Definition 4.15 (Domain Consistency):

$$c : \langle \mathsf{E}', l' \rangle \in \Sigma \quad \leftrightarrow \quad c : l' \in \Delta_1$$

$$c : \langle \mathsf{E}', l' \rangle \in \Sigma \quad \leftrightarrow \quad c : l' \in \Delta_2$$

$$k : \langle \mathsf{E}', l' \rangle \in \Sigma \quad \leftrightarrow \quad k : l' \in \Delta_1$$

$$k : \langle \mathsf{E}', l' \rangle \in \Sigma \quad \leftrightarrow \quad k : l' \in \Delta_2$$

$$a : \langle \langle \mathsf{E}', l' \rangle \, \mathsf{loc}, l'' \rangle \in \Sigma \quad \leftrightarrow \quad a : l' \in \Delta_1, (a \mapsto v_1) \in \Delta_1 \quad \text{and} \quad \vdash_\perp v_1 : \langle \mathsf{E}', l' \rangle$$

$$a : \langle \langle \mathsf{E}', l' \rangle \, \mathsf{loc}, l'' \rangle \in \Sigma \quad \leftrightarrow \quad a : l' \in \Delta_2, (a \mapsto v_2) \in \Delta_2 \quad \text{and} \quad \vdash_\perp v_2 : \langle \mathsf{E}', l' \rangle$$

Consequently:

$$\left.\begin{array}{l} c : l' \in \Delta_1 \;\leftrightarrow\; c : l' \in \Delta_2 \\ k : l' \in \Delta_1 \;\leftrightarrow\; k : l' \in \Delta_2 \\ a : l' \in \Delta_1 \;\leftrightarrow\; a : l' \in \Delta_2 \end{array}\right\} \Rightarrow \; \mathbb{L}_{\Delta_1} = \mathbb{L}_{\Delta_2} = \mathbb{L}$$

$$(a \mapsto v_1) \in \Delta_1 \;\leftrightarrow\; (a \mapsto v_2) \in \Delta_2 \quad \Rightarrow \quad \mathsf{dom}(\phi_{\Delta_1}) = \mathsf{dom}(\phi_{\Delta_2})$$

By Lemma 4.22 (Indistinguishability):

$$\forall a : l' \in \mathbb{L}, l' \leq l \rightarrow \mathbb{L} \vdash \phi_{\Delta_1}[a] \approx_l \phi_{\Delta_2}[a]$$

The result therefore follows from Definition 4.9 (Indistinguishability of Evaluation Environments). □

A.2.3 Theorems

Theorem 4.21 (Non-Interference for Typed Expressions)

If $\Gamma \vdash_{l_w} e : S$ *w.r.t.* Σ, $\Pi \vdash \langle \Delta, e\rangle \Downarrow \langle \Delta_r, v\rangle$ *and* $\Pi' \vdash \langle \Delta', e\rangle \Downarrow \langle \Delta'_r, v'\rangle$, *where* $(\Gamma, \Sigma) \simeq (\Pi, \Delta)$, $(\Gamma, \Sigma) \simeq (\Pi', \Delta')$, Σ *is valid,* $\Gamma \vdash \Pi \approx_{l_o} \Pi'$, $\Sigma \vdash \Delta \approx_{l_o} \Delta'$ *and* $l_o \leq l_w$, *then* $v \approx_{l_o} v' : S$ *and* $\Sigma \vdash \Delta_r \approx_{l_o} \Delta'_r$.

Proof. We proceed by induction on the structure of the derivation of $\Gamma \vdash_{l_w} e : \mathsf{S}$ *w.r.t.* Σ.

- Cases T-CNST, T-LOC, T-KEY, T-CTXT, T-TRUE and T-FALSE:

  Here, e is a value therefore, by inspection of the evaluation rules, only E-VAL could have been used at the root of each evaluation, thus:

  $v = e$

  $v' = e$

  By the first rule in Definition A.6 (Typed Indistinguishability of Values):

  $\underline{v \approx_{l_w} v' : \mathsf{S}}$

- Case T-SENC:

  We have:

  $e = \mathrm{senc}(e_k, e_m)$

  $\mathsf{S} = \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m\rangle), \bot\rangle$

  $\Gamma \vdash_{l_w} e_k : \langle l'\ l_m\ \mathsf{key}, \top\rangle$

  $\Gamma \vdash_{l_w} e_m : \langle \mathsf{E}_m, l_m\rangle$

  $l' \leq \lfloor \langle \mathsf{E}_m, l_m\rangle \rfloor$

  By inspection of the evaluation rules, only E-SENC could have been used at the root of each evaluation, therefore we must have:

  $$\begin{array}{lcl} \Pi \vdash \langle \Delta, e_k\rangle \Downarrow \langle \Delta_1, k\rangle & & \Pi' \vdash \langle \Delta', e_k\rangle \Downarrow \langle \Delta'_1, k'\rangle \\ \Pi \vdash \langle \Delta_1, e_m\rangle \Downarrow \langle \Delta_2, v_m\rangle & & \Pi' \vdash \langle \Delta'_1, e_m\rangle \Downarrow \langle \Delta'_2, v'_m\rangle \\ v = \mathrm{ctxt}(n, k, v_m) & \text{and} & v' = \mathrm{ctxt}(n', k', v'_m) \\ \Delta_r = \Delta_2 \cup n & & \Delta'_r = \Delta'_2 \cup n' \\ n \notin \Delta_2 & & n' \notin \Delta'_2 \end{array}$$

  By multiple applications of Theorem 4.18 (Type Safety):

  $$\begin{array}{ll} (\Gamma, \Sigma) \simeq (\Pi, \Delta_1) & (\Gamma, \Sigma) \simeq (\Pi', \Delta'_1) \\ (\Gamma, \Sigma) \simeq (\Pi, \Delta_2) & (\Gamma, \Sigma) \simeq (\Pi', \Delta'_2) \end{array}$$

By two applications of the induction hypothesis, we get:

$$k \approx_{l_o} k' : \langle l'\ l_m\ \mathsf{key}, \top \rangle \qquad v_m \approx_{l_o} v'_m : \langle \mathsf{E}_m, l_m \rangle$$
$$\Sigma \vdash \Delta_1 \approx_{l_o} \Delta'_1 \quad \text{and} \quad \Sigma \vdash \Delta_2 \approx_{l_o} \Delta'_2$$

By the fifth rule in Definition A.6 (Typed Indistinguishability of Values):

$$v \approx_{l_o} v' : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \bot \rangle \quad \equiv \quad \underline{v \approx_{l_o} v' : \mathsf{S}}$$

By Definition A.7 (Typed Indistinguishability of Evaluation Environments):

$$\Sigma \vdash (\Delta_2 \cup n) \approx_{l_o} (\Delta'_2 \cup n') \quad \equiv \quad \underline{\Sigma \vdash \Delta_r \approx_{l_o} \Delta'_r}$$

- Case T-SDEC:

  We have:

  $$e \ = \ \text{try sdec}(e_k, e_c) = x \text{ in } e_1 \text{ else } e_2$$
  $$\Gamma \vdash_{l_w} e_k : \langle l'\ l_m\ \mathsf{key}, \top \rangle$$
  $$\Gamma \vdash_{l_w} e_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle$$
  $$l' \leq \lfloor \langle \mathsf{E}_m, l_m \rangle \rfloor$$
  $$x \notin \mathsf{dom}(\Sigma)$$
  $$(\Gamma, x : \langle \mathsf{E}_m, l_m \rangle) \vdash_{l' \curlyvee l_w} e_1 : \mathsf{S}$$
  $$\Gamma \vdash_{l' \curlyvee l_w} e_2 : \mathsf{S}$$

  By inspection of the evaluation rules, either E-SDEC1 or E-SDEC2 could have been used at the root of each evaluation. In both cases, the key and ciphertext expressions are evaluated in that order, therefore we must have:

  $$\Pi \vdash \langle \Delta, e_k \rangle \Downarrow \langle \Delta_1, k \rangle \qquad \Pi' \vdash \langle \Delta', e_k \rangle \Downarrow \langle \Delta'_1, k' \rangle$$
  $$\Pi \vdash \langle \Delta_1, e_c \rangle \Downarrow \langle \Delta_2, v_c \rangle \quad \text{and} \quad \Pi' \vdash \langle \Delta'_1, e_c \rangle \Downarrow \langle \Delta'_2, v'_c \rangle$$

  By multiple applications of Theorem 4.18 (Type Safety):

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1) \qquad (\Gamma, \Sigma) \simeq (\Pi', \Delta'_1)$$
  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_2) \qquad (\Gamma, \Sigma) \simeq (\Pi', \Delta'_2)$$
  $$\vdash_{l_w} v_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle \qquad \vdash_{l_w} v'_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), \top \rangle$$

  By two applications of the induction hypothesis, we get:

  $$k \approx_{l_o} k' : \langle l'\ l_m\ \mathsf{key}, l \rangle \qquad v_c \approx_{l_o} v'_c : \langle \mathsf{enc}(\langle \mathsf{E}_m, l_m \rangle), l \rangle$$
  $$\Sigma \vdash \Delta_1 \approx_{l_o} \Delta'_1 \quad \text{and} \quad \Sigma \vdash \Delta_2 \approx_{l_o} \Delta'_2$$

By Lemma A.3 (Canonical Forms):

$v_c = \text{ctxt}(n, k_1, v_m)$

$v'_c = \text{ctxt}(n', k_2, v'_m)$

By Definition A.6 (Typed Indistinguishability of Values):

$k = k' \quad or \quad l_o < l'$

$k_1 = k_2 \quad or \quad l_o < l'$

We now split on the relative ordering of l_o and l':

- Subcase $l_o < l'$:

  By the properties of $\curlyvee$:

  $l_o < (l' \curlyvee l_w)$

  By Theorem 4.18 (Type Safety):

  $\vdash_{l_w} k : \langle l'\ l_m\ \mathsf{key}, \top \rangle$

  In this subcase, each evaluation may have E-SDEC1 or E-SDEC2 at its root, giving us:

  $$\begin{array}{ll} & \Pi \cup (x \mapsto v_m) \vdash \langle \Delta_2, e_1 \rangle \Downarrow \langle \Delta_r, v_1 \rangle \\ v \in \{v_1, v_2\} & \Pi \vdash \langle \Delta_2, e_2 \rangle \Downarrow \langle \Delta_r, v_2 \rangle \\ v' \in \{v'_1, v'_2\} \quad \text{where} & \Pi' \cup (x \mapsto v'_m) \vdash \langle \Delta'_2, e_1 \rangle \Downarrow \langle \Delta'_r, v_2 \rangle \\ & \Pi' \vdash \langle \Delta'_2, e_2 \rangle \Downarrow \langle \Delta'_r, v'_2 \rangle \end{array}$$

  By Definition 4.15 (Domain Consistency):

  $((\Gamma, x : \langle \mathsf{E}_m, l_m \rangle), \Sigma) \simeq (\Pi \cup (x \mapsto v_m), \Delta_2)$

  $((\Gamma, x : \langle \mathsf{E}_m, l_m \rangle), \Sigma) \simeq (\Pi' \cup (x \mapsto v'_m), \Delta'_2)$

  By two applications of Theorem 4.18 (Type Safety):

  $\vdash_{l' \curlyvee l_w} v : \mathsf{S}$

  $\vdash_{l' \curlyvee l_w} v' : \mathsf{S}$

  By Lemma 4.24 (Typed Indistinguishability of Arbitrary Values):

  $\underline{v \approx_{l_o} v' : \mathsf{S}}$

  By two separate applications of Lemma A.12 (Preservation Under Evaluation of Typed Indistinguishability for Evaluation Environments):

  $\Sigma \vdash \Delta_2 \approx_{l_o} \Delta_r$

  $\Sigma \vdash \Delta'_2 \approx_{l_o} \Delta'_r$

By two applications of Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

$$\underline{\Sigma \vdash \Delta_r \approx_{l_o} \Delta'_r}$$

- Subcase $l_o \not\preceq l'$:

  In this subcase, we must have:

  $$\left.\begin{array}{l} k = k' \\ k_1 = k_2 \end{array}\right\} \Rightarrow \quad k = k_1 \text{ iff } k' = k_2$$

  As a result, both evaluations must have the same rule at their root, giving us:

  $$\begin{array}{ll} \Pi \cup (x \mapsto v_m) \vdash \langle \Delta_2, e_1 \rangle \Downarrow \langle \Delta_r, v \rangle & \Pi \vdash \langle \Delta_2, e_2 \rangle \Downarrow \langle \Delta_r, v \rangle \\ \Pi' \cup (x \mapsto v'_m) \vdash \langle \Delta'_2, e_1 \rangle \Downarrow \langle \Delta'_r, v' \rangle & \Pi' \vdash \langle \Delta'_2, e_2 \rangle \Downarrow \langle \Delta'_r, v' \rangle \end{array} \quad \textit{or}$$

  By Definition A.8 (Typed Indistinguishability of Evaluation Contexts):

  $$(\Gamma, x : \langle \mathsf{E}_m, l_m \rangle) \vdash \Pi \cup (x \mapsto v_m) \approx_{l_o} \Pi' \cup (x \mapsto v'_m)$$

  Consequently, for each of the two possible outcomes, the result follows from the induction hypothesis.

- Case T-IF:

  We have:

  $$e \;=\; \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

  $$\Gamma \vdash_{l_w} e_1 : \langle \mathsf{bool}, l \rangle$$

  $$\Gamma \vdash_{l \curlyvee l_w} e_2 : \mathsf{S}$$

  $$\Gamma \vdash_{l \curlyvee l_w} e_3 : \mathsf{S}$$

  By inspection of the evaluation rules, either E-IF1 or E-IF2 could have been used at the root of each evaluation. In both cases, the guard expression will have been evaluated first, therefore we must have:

  $$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta_1, v_1 \rangle$$

  $$\Pi' \vdash \langle \Delta', e_1 \rangle \Downarrow \langle \Delta'_1, v'_1 \rangle$$

  By the induction hypothesis:

  $$v_1 \approx_{l_o} v'_1 : \langle \mathsf{bool}, l \rangle$$

  $$\Sigma \vdash \Delta_1 \approx_{l_o} \Delta'_1$$

  By two applications of Theorem 4.18 (Type Safety):

  $$\begin{array}{ll} \vdash_{l_w} v_1 : \langle \mathsf{bool}, l \rangle & \vdash_{l_w} v'_1 : \langle \mathsf{bool}, l \rangle \\ (\Gamma, \Sigma) \simeq (\Pi, \Delta_1) & (\Gamma, \Sigma) \simeq (\Pi', \Delta'_1) \end{array} \quad \text{and}$$

By Lemma A.3 (Canonical Forms):

$v_1 \in \{\text{TRUE}, \text{FALSE}\}$

$v'_1 \in \{\text{TRUE}, \text{FALSE}\}$

We now split on the relative ordering of l_o and l:

- Subcase $l_o < l$:

  Here, each evaluation may have E-IF1 or E-IF2 at its root, thus:

  $$\begin{array}{ll} v \in \{v_2, v_3\} & \\ v' \in \{v'_2, v'_3\} & \text{where} \end{array} \quad \begin{array}{l} \Pi \vdash \langle \Delta_1, e_2 \rangle \Downarrow \langle \Delta_r, v_2 \rangle \\ \Pi \vdash \langle \Delta_1, e_3 \rangle \Downarrow \langle \Delta_r, v_3 \rangle \\ \Pi' \vdash \langle \Delta'_1, e_2 \rangle \Downarrow \langle \Delta'_r, v'_2 \rangle \\ \Pi' \vdash \langle \Delta'_1, e_3 \rangle \Downarrow \langle \Delta'_r, v'_3 \rangle \end{array}$$

  In all outcomes, Theorem 4.18 (Type Safety) gives us:

  $\vdash_l v : \mathsf{S}$

  $\vdash_l v' : \mathsf{S}$

  By Lemma 4.24 (Typed Indistinguishability of Arbitrary Values):

  $\underline{v \approx_{l_o} v' : \mathsf{S}}$

  By the properties of Υ:

  $l \leq (l \Upsilon l_w) \qquad \Rightarrow \qquad l_o < (l \Upsilon l_w)$

  By two separate applications of Lemma A.12 (Preservation Under Evaluation of Typed Indistinguishability for Evaluation Environments):

  $\Sigma \vdash \Delta_1 \approx_{l_o} \Delta_r$

  $\Sigma \vdash \Delta'_1 \approx_{l_o} \Delta'_r$

  By two applications of Lemma A.11 (Transitivity of Typed Indistinguishability for Evaluation Environments):

  $\underline{\Sigma \vdash \Delta_r \approx_{l_o} \Delta'_r}$

- Subcase $l_o \not< l$:

  In this subcase, it follows from Definition A.6 (Typed Indistinguishability of Values) that:

  $v_1 = v'_1$

  As a result, both evaluations must have the same rule at their root, giving us:

  $$\begin{array}{ll} \Pi \vdash \langle \Delta_1, e_2 \rangle \Downarrow \langle \Delta_r, v \rangle & \\ \Pi' \vdash \langle \Delta'_1, e_2 \rangle \Downarrow \langle \Delta'_r, v' \rangle & \textit{or} \end{array} \quad \begin{array}{l} \Pi \vdash \langle \Delta_1, e_3 \rangle \Downarrow \langle \Delta_r, v \rangle \\ \Pi' \vdash \langle \Delta'_3, e_3 \rangle \Downarrow \langle \Delta'_r, v' \rangle \end{array}$$

In each possibility, the result follows from the induction hypothesis.

- Case T-LET:

  We have:

  $e = \text{let } x = e_1 \text{ in } e_2$

  $\Gamma \vdash_{l_w} e_1 : \mathsf{S}'$

  $x \notin \mathsf{dom}(\Sigma)$

  $(\Gamma, x : \mathsf{S}') \vdash_{l_w} e_2 : \mathsf{S}$

  By inspection of the evaluation rules, only E-LET could have been used at the root of each evaluation, therefore we must have:

  $\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta_1, v_1 \rangle$

  $\Pi \cup (x \mapsto v_1) \vdash \langle \Delta_1, e_2 \rangle \Downarrow \langle \Delta_r, v \rangle$

  $\Pi' \vdash \langle \Delta', e_1 \rangle \Downarrow \langle \Delta'_1, v'_1 \rangle$

  $\Pi' \cup (x \mapsto v'_1) \vdash \langle \Delta'_1, e_2 \rangle \Downarrow \langle \Delta'_r, v' \rangle$

  By the induction hypothesis:

  $v_1 \approx_{l_o} v'_1 : \mathsf{S}'$

  $\Sigma \vdash \Delta_1 \approx_{l_o} \Delta'_1$

  By Theorem 4.18 (Type Safety):

  $\vdash_{l_w} v_1 : \mathsf{S}'$

  $\vdash_{l_w} v'_1 : \mathsf{S}'$

  $(\Gamma, \Sigma) \simeq (\Pi, \Delta_1)$

  $(\Gamma, \Sigma) \simeq (\Pi', \Delta'_1)$

  By Definition 4.15 (Domain Consistency):

  $((\Gamma, x : \mathsf{S}'), \Sigma) \simeq (\Pi \cup (x \mapsto v_1), \Delta_1)$

  $((\Gamma, x : \mathsf{S}'), \Sigma) \simeq (\Pi' \cup (x \mapsto v'_1), \Delta'_1)$

  By Definition A.8 (Typed Indistinguishability of Evaluation Contexts):

  $(\Gamma, x : \mathsf{S}') \vdash \Pi \cup (x \mapsto v_1) \approx_{l_o} \Pi' \cup (x \mapsto v'_1)$

  Consequently, the result follows from the induction hypothesis.

- T-EQ

  We have:

  $$e = e_1 == e_2$$
  $$\mathsf{S} = \langle \mathsf{bool}, l \rangle$$
  $$\Gamma \vdash_{l_w} e_1 : \langle \mathsf{E}, l \rangle$$
  $$\Gamma \vdash_{l_w} e_2 : \langle \mathsf{E}, l \rangle$$
  $$\mathsf{E} \not\equiv \mathsf{enc(S)}$$

  By inspection of the evaluation rules, either E-EQ1 or E-EQ2 could have been used at the root of each evaluation. In both cases, the two sub-expressions will have been evaluated in left-to-right order, therefore we must have:

  $$\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta_1, v_1 \rangle$$
  $$\Pi \vdash \langle \Delta_1, e_2 \rangle \Downarrow \langle \Delta_r, v_2 \rangle$$
  $$\Pi' \vdash \langle \Delta', e_1 \rangle \Downarrow \langle \Delta'_1, v'_1 \rangle$$
  $$\Pi' \vdash \langle \Delta'_1, e_2 \rangle \Downarrow \langle \Delta'_r, v'_2 \rangle$$

  By Theorem 4.18 (Type Safety):

  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_1) \quad \text{and} \quad (\Gamma, \Sigma) \simeq (\Pi', \Delta'_1)$$
  $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_r) \quad \text{and} \quad (\Gamma, \Sigma) \simeq (\Pi', \Delta'_r)$$

  By two applications of the induction hypothesis:

  $$v_1 \approx_{l_o} v'_1 : \langle \mathsf{E}, l \rangle \quad \text{then} \quad v_2 \approx_{l_o} v'_2 : \langle \mathsf{E}, l \rangle$$
  $$\Sigma \vdash \Delta_1 \approx_{l_o} \Delta'_1 \quad \text{then} \quad \underline{\Sigma \vdash \Delta_r \approx_{l_o} \Delta'_r}$$

  We now split on the relative ordering of l_o and l:

  - Subcase $l_o < l$:

    Here, each evaluation may have E-EQ1 or E-EQ2 at its root, giving us:

    $$v \in \{\text{TRUE}, \text{FALSE}\} \quad \text{where} \quad \Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta_r, v \rangle$$
    $$v' \in \{\text{TRUE}, \text{FALSE}\} \quad \text{where} \quad \Pi' \vdash \langle \Delta', e \rangle \Downarrow \langle \Delta'_r, v' \rangle$$

    Thus, for all possible outcomes, the other part of the result follows from Definition A.6 (Typed Indistinguishability of Values).

- Subcase $l_o \not\preceq l$:

  By Theorem 4.18 (Type Safety):

  $$\vdash_{l_w} v_1 : \langle \mathsf{E}, l \rangle$$
  $$\vdash_{l_w} v_2 : \langle \mathsf{E}, l \rangle$$
  $$\vdash_{l_w} v_1' : \langle \mathsf{E}, l \rangle$$
  $$\vdash_{l_w} v_2' : \langle \mathsf{E}, l \rangle$$

  By Lemma A.3 (Canonical Forms):

$\mathsf{E} = l'\ l_m\ \mathsf{key}$		$\mathsf{E} = \mathsf{data}$		$\mathsf{E} = \mathsf{bool}$
$v_1 = k_1$		$v_1 = c_1$		$v_1 \in \{\text{TRUE}, \text{FALSE}\}$
$v_2 = k_2$	*or*	$v_2 = c_2$	*or*	$v_2 \in \{\text{TRUE}, \text{FALSE}\}$
$v_1' = k_1'$		$v_1' = c_1'$		$v_1' \in \{\text{TRUE}, \text{FALSE}\}$
$v_2' = k_1'$		$v_2' = c_2'$		$v_2' \in \{\text{TRUE}, \text{FALSE}\}$

  If $\mathsf{E} = l'\ l_m\ \mathsf{key}$, by Lemma A.2 (Inversion of the Typing Relation) we get:

  $$l' \leq l \quad \Rightarrow \quad l_o \not\preceq l'$$

  Consequently, for all possible forms of E, it follows from Definition A.6 (Typed Indistinguishability of Values) that:

  $$\left.\begin{array}{l} v_1 = v_1' \\ v_2 = v_2' \end{array}\right\} \Rightarrow \quad v_1 = v_2 \text{ iff } v_1' = v_2'$$

  As a result, both evaluations must have the same rule at their root, giving us:

$\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta_r, \text{TRUE} \rangle$		$\Pi \vdash \langle \Delta, e \rangle \Downarrow \langle \Delta_r, \text{FALSE} \rangle$
$\Pi' \vdash \langle \Delta', e \rangle \Downarrow \langle \Delta_r', \text{TRUE} \rangle$	*or*	$\Pi' \vdash \langle \Delta', e \rangle \Downarrow \langle \Delta_r', \text{FALSE} \rangle$

  The other part of the result follows via Definition A.6 (Typed Indistinguishability of Values).

- Case T-VAR:

  We have:

  $$e = x$$
  $$\mathsf{S} = \langle \mathsf{E}, l \curlyvee l_w \rangle$$
  $$\Gamma = [x : \langle \mathsf{E}, l \rangle]$$
  $$l_w \leq \lfloor \langle \mathsf{E}, l \curlyvee l_w \rangle \rfloor$$

By inspection of the evaluation rules, only E-VAR could have been used at the root of each evaluation, therefore we must have:

$$(x \mapsto v) \in \Pi$$
$$(x \mapsto v') \in \Pi'$$
$$\left.\begin{array}{l}\Delta_r = \Delta \\ \Delta'_r = \Delta'\end{array}\right\} \Rightarrow \underline{\Sigma \vdash \Delta_r \approx_{l_o} \Delta'_r}$$

By Definition A.8 (Typed Indistinguishability of Evaluation Contexts):

$$v \approx_{l_o} v' : \langle \mathsf{E}, l \rangle$$

By the properties of Υ:

$$l \leq (l \Upsilon l_w) \quad \Rightarrow \quad \langle \mathsf{E}, l \rangle <: \mathsf{S}$$

The result then follows via Lemma A.9 (Widening of Typed Indistinguishability for Values).

- Case T-ASGN:

  We have:

  $$e \;=\; e_1 := e_2$$
  $$\Gamma \vdash_{l_w} e_1 : \langle \mathsf{S\,loc}, \top \rangle$$
  $$\Gamma \vdash_{l_w} e_2 : \mathsf{S}$$

  By inspection of the evaluation rules, only E-ASGN could have been used at the root of each evaluation, therefore we must have:

  $$\begin{array}{ll}\Pi \vdash \langle \Delta, e_1 \rangle \Downarrow \langle \Delta_1, a \rangle & \\ \Pi \vdash \langle \Delta_1, e_2 \rangle \Downarrow \langle \Delta_2, v \rangle & \end{array} \quad \text{and} \quad \begin{array}{l}\Pi' \vdash \langle \Delta', e_1 \rangle \Downarrow \langle \Delta'_1, a' \rangle \\ \Pi' \vdash \langle \Delta'_1, e_2 \rangle \Downarrow \langle \Delta'_2, v' \rangle\end{array}$$

  By Theorem 4.18 (Type Safety):

  $$\begin{array}{ll}(\Gamma, \Sigma) \simeq (\Pi, \Delta_1) & \\ (\Gamma, \Sigma) \simeq (\Pi, \Delta_2) & \end{array} \quad \text{and} \quad \begin{array}{l}(\Gamma, \Sigma) \simeq (\Pi', \Delta'_1) \\ (\Gamma, \Sigma) \simeq (\Pi', \Delta'_2)\end{array}$$

  By the induction hypothesis:

  $$\Sigma \vdash \Delta_1 \approx_{l_o} \Delta'_1$$
  $$a \approx_{l_o} a' : \langle \mathsf{S\,loc}, \top \rangle$$

  By a second application of the induction hypothesis:

  $$\Sigma \vdash \Delta_2 \approx_{l_o} \Delta'_2$$
  $$\underline{v \approx_{l_o} v' : \mathsf{S}}$$

We now split of the equality or otherwise of a and a':

- Subcase $a = a'$:

  By Definition A.7 (Typed Indistinguishability of Evaluation Environments):

  $$\Sigma \vdash \Delta_2(a \mapsto v) \approx_{l_o} \Delta_2'(a' \mapsto v') \qquad \equiv \qquad \underline{\Sigma \vdash \Delta_r \approx_{l_o} \Delta_r'}$$

- Subcase $a \neq a'$:

  By inspection of Definition A.6 (Typed Indistinguishability of Values), we must have:

  $$\forall v_1, v_2 \text{ s.t. } \vdash_\perp v_1 : \mathsf{S} \text{ and } \vdash_\perp v_2 : \mathsf{S},\ v_1 \approx_{l_o} v_2 : \mathsf{S}$$

  By Theorem 4.18 (Type Safety):

  $$\vdash_{l_w} a : \langle \mathsf{S\ loc}, \top \rangle$$
  $$\vdash_{l_w} a' : \langle \mathsf{S\ loc}, \top \rangle$$
  $$\vdash_{l_w} v : \mathsf{S}$$
  $$\vdash_{l_w} v' : \mathsf{S}$$

  By Lemma A.2 (Inversion of the Typing Relation):

  $$a : \langle \mathsf{S\ loc}, l_1 \rangle \in \Sigma$$
  $$a' : \langle \mathsf{S\ loc}, l_2 \rangle \in \Sigma$$

  By Definition 4.15 (Domain Consistency):

  $$\vdash_\perp \Delta_2[a] : \mathsf{S}$$
  $$\vdash_\perp \Delta_2[a'] : \mathsf{S}$$
  $$\vdash_\perp \Delta_2'[a] : \mathsf{S}$$
  $$\vdash_\perp \Delta_2'[a'] : \mathsf{S}$$

  By T-SUB and S-REFL:

  $$\vdash_\perp v : \mathsf{S}$$
  $$\vdash_\perp v' : \mathsf{S}$$

  Consequently, from the preconditions of this subcase, we get:

  $$\left.\begin{array}{l} v \approx_{l_o} \Delta_2'[a] : \mathsf{S} \\ \Delta_2[a'] \approx_{l_o} v' : \mathsf{S} \end{array}\right\} \Rightarrow \quad \underline{\Sigma \vdash \Delta_r \approx_{l_o} \Delta_r'}$$

- Case T-DREF:

  We have:

  $$e = *e'$$
  $$\mathsf{S} = \langle \mathsf{E}, l \curlyvee l_w \rangle$$
  $$\Gamma \vdash_{l_w} e' : \langle \langle \mathsf{E}, l \rangle \, \mathsf{loc}, \top \rangle$$
  $$l_w \leq \lfloor \mathsf{S} \rfloor$$

  By inspection of the evaluation rules, only E-DREF could have been used at the root of each evaluation, therefore we must have:

  $$\Pi \vdash \langle \Delta, e' \rangle \Downarrow \langle \Delta_r, a \rangle \qquad \text{and} \qquad \Pi' \vdash \langle \Delta', e' \rangle \Downarrow \langle \Delta'_r, a' \rangle$$
  $$v = \Delta_r(a) \qquad\qquad v' = \Delta'_r(a')$$

  By the induction hypothesis:

  $$a \approx_{l_o} a' : \langle \langle \mathsf{E}, l \rangle \, \mathsf{loc}, \top \rangle$$
  $$\underline{\Sigma \vdash \Delta_r \approx_{l_o} \Delta'_r}$$

  We now split of the equality or otherwise of a and a':

  - Subcase $a = a'$:

    By Definition A.7 (Typed Indistinguishability of Evaluation Environments):

    $$\Delta_r(a) \approx_{l_o} \Delta'_r(a') : \langle \mathsf{E}, l \rangle \qquad \equiv \qquad v \approx_{l_o} v' : \langle \mathsf{E}, l \rangle$$

    By the properties of $\curlyvee$:

    $$l \leq (l \curlyvee l_w) \qquad \Rightarrow \qquad \langle \mathsf{E}, l \rangle <: \mathsf{S}$$

    By Lemma A.9 (Widening of Typed Indistinguishability for Values):

    $$\underline{v \approx_{l_o} v' : \mathsf{S}}$$

  - Subcase $a \neq a'$:

    By inspection of Definition A.6 (Typed Indistinguishability of Values), we must have:

    $$l_o < \top$$
    $$\forall v_1, v_2 \text{ s.t. } \vdash_{\perp} v_1 : \langle \mathsf{E}, l \rangle \text{ and } \vdash_{\perp} v_2 : \langle \mathsf{E}, l \rangle, \, v_1 \approx_{l_o} v_2 : \langle \mathsf{E}, l \rangle$$

    By Theorem 4.18 (Type Safety):

    $$\vdash_{l_w} a : \langle \langle \mathsf{E}, l \rangle \, \mathsf{loc}, \top \rangle \qquad \text{and} \qquad \vdash_{l_w} a' : \langle \langle \mathsf{E}, l \rangle \, \mathsf{loc}, \top \rangle$$
    $$(\Gamma, \Sigma) \simeq (\Pi, \Delta_r) \qquad\qquad (\Gamma, \Sigma) \simeq (\Pi', \Delta'_r)$$

By Lemma A.2 (Inversion of the Typing Relation):

$a : \langle \langle \mathsf{E}, l \rangle \, \mathsf{loc}, l_1 \rangle \in \Sigma$

$a' : \langle \langle \mathsf{E}, l \rangle \, \mathsf{loc}, l_2 \rangle \in \Sigma$

By Definition 4.15 (Domain Consistency):

$\vdash_{\perp} \Delta_r(a) : \langle \mathsf{E}, l \rangle$

$\vdash_{\perp} \Delta'_r(a') : \langle \mathsf{E}, l \rangle$

Taking $v_1 = \Delta_r(a)$ and $v_2 = \Delta'_r(a')$, we get:

$\Delta_r(a) \approx_{l_o} \Delta'_r(a') : \langle \mathsf{E}, l \rangle \qquad \equiv \qquad v \approx_{l_o} v' : \langle \mathsf{E}, l \rangle$

By the properties of Υ:

$l \leq (l \Upsilon l_w) \quad \Rightarrow \quad \langle \mathsf{E}, l \rangle <: \mathsf{S}$

By Lemma A.9 (Widening of Typed Indistinguishability for Values):

$\underline{v \approx_{l_o} v' : \mathsf{S}}$

- T-FnApp:

  We have:

  $e \;=\; f(e_i \,^{i \in 1..n})$

  $f : \{ i : \mathsf{S}_i \,^{i \in 1..n} \} \xrightarrow{l'_w} \langle \mathsf{E}, l \rangle \in \Sigma$

  $l_w \leq l'_w$

  $\forall i \in 1..n \;\; \Gamma \vdash_{l_w} e_i : \mathsf{S}_i$

  By inspection of the evaluation rules, only E-FnApp could have been used at the root of each evaluation, therefore we must have:

  $\forall i \in 1..n \;\; \Pi \vdash \langle \Delta_{i-1}, e_i \rangle \Downarrow \langle \Delta_i, v_i \rangle$

  $f(x_i \,^{i \in 1..n}) = \; e_f \in \Delta_n$

  $(\Pi \cup x''_i \mapsto v_i) \vdash \langle \Delta_n, [x''_i / x_i] e_f \rangle \Downarrow \langle \Delta_r, v \rangle$

  and

  $\forall i \in 1..n \;\; \Pi' \vdash \langle \Delta'_{i-1}, e_i \rangle \Downarrow \langle \Delta'_i, v'_i \rangle$

  $f(x'_i \,^{i \in 1..n}) = \; e'_f \in \Delta'_n$

  $(\Pi' \cup x''_i \mapsto v'_i) \vdash \langle \Delta'_n, [x''_i / x'_i] e'_f \rangle \Downarrow \langle \Delta'_r, v' \rangle$

  where

$x_i'' \notin \mathsf{dom}(\Pi)$

$x_i'' \notin \mathsf{dom}(\Pi')$

$x_i'' \notin FV(e_f)$

$x_i'' \notin FV(e_f')$

By n applications of Theorem 4.18 (Type Safety):

$$\left.\begin{array}{l} \vdash_{l_w} v_i : \mathsf{S}_i \\ (\Gamma, \Sigma) \simeq (\Pi, \Delta_i) \end{array}\right. \quad \text{and} \quad \left.\begin{array}{l} \vdash_{l_w} v_i' : \mathsf{S}_i \\ (\Gamma, \Sigma) \simeq (\Pi', \Delta_i') \end{array}\right\} \ \forall i \in 1..n$$

By n applications of the induction hypothesis:

$$\left.\begin{array}{l} v_i \approx_{l_o} v_i' : \mathsf{S}_i \\ \Sigma \vdash \Delta_i \approx_{l_o} \Delta_i' \end{array}\right\} \ \forall i \in 1..n$$

By Definition A.7 (Typed Indistinguishability of Evaluation Environments):

$$\left.\begin{array}{l} e_f \ = \ e_f' \\ \forall i \in 1..n \ \ x_i = x_i' \end{array}\right\} \ \Rightarrow \ [x_i''/x_i]e_f \ = \ [x_i''/x_i']e_f'$$

By T-SUB and S-REFL:

$$\left.\begin{array}{l} \vdash_{\perp} v_i : \mathsf{S}_i \ \textit{w.r.t.} \ \Sigma \\ \vdash_{\perp} v_i' : \mathsf{S}_i \ \textit{w.r.t.} \ \Sigma \end{array}\right\} \ \forall i \in 1..n$$

By Definition 4.15 (Domain Consistency):

$x_i'' \notin \mathsf{dom}(\Gamma)$

$(\Gamma, x_i : \mathsf{S}_i) \vdash_{l_w'} e_f : \mathsf{S} \qquad \Rightarrow \qquad (\Gamma, x_i'' : \mathsf{S}_i) \vdash_{l_w'} [x_i''/x_i]e_f : \mathsf{S}$

$((\Gamma, x_i'' : \mathsf{S}_i), \Sigma) \simeq ((\Pi \cup x_i'' \mapsto v_i), \Delta_n)$

$((\Gamma, x_i'' : \mathsf{S}_i), \Sigma) \simeq ((\Pi' \cup x_i'' \mapsto v_i'), \Delta_n')$

By the transitivity of $\leq$:

$l_o \leq l_w'$

The result therefore follows from the induction hypothesis.

- T-SUB:

  We have:

  $\Gamma' \vdash_{l_w'} e : \mathsf{S}'$

  $\Gamma' \subseteq \Gamma$

  $\mathsf{S}' <: \mathsf{S}$

  $l_w \leq l_w' \qquad \Rightarrow \qquad l_o \leq l_w'$

By the induction hypothesis:

$\Gamma' \vdash \Delta_r \approx_{l_o} \Delta'_r$

$v \approx_{l_o} v' : \mathsf{S}'$

By inspection of Definition A.7 (Typed Indistinguishability of Evaluation Environments):

$\underline{\Gamma \vdash \Delta_r \approx_{l_o} \Delta'_r}$

By Lemma A.9 (Widening of Typed Indistinguishability for Values):

$\underline{v \approx_{l_o} v' : \mathsf{S}}$